Osnovne podatkovne strukture in algoritmi

Janez Brank,
Andrej Brodnik

Kazalo

- Reference (kazalci)
- Seznami (linked lists)
- Vrste, skladi
- Razpršene tabele (hash tables)
- Urejanje
- Iskanje *k*-tega najmanjšega elementa
- Bisekcija
- Kopica
- Binarno iskalno drevo
- Razširjeno binarno iskalno drevo

Reference

- Referenca (kazalec, *pointer*) je spremenljivka, ki hrani naslov nečesa v pomnilniku.
  - Ponavadi ob deklaraciji kazalca tudi povemo, da bo kazal na objekt točno določenega tipa:

```
int *p; // naj bi kazal na int
typedef struct { int bar, baz; } Foo;
Foo *r; // naj bi kazal na primerek strukture Foo
void* q; // referenca na karkoli
```

  - Z operatorjem & (v C/C++) ali @ (v pascalu) lahko dobimo naslov nekega že obstoječega objekta. Z njim lahko inicializiramo referenco (kazalec).
  - Z operatorjem * (v C/C++) ali ^ (v pascalu) referenco dereferenciramo lahko dostopamo do vrednosti, na katero kaže referenca.

```
int i = 10, j = 20;
int *pi = &i, *pj = &j;
printf("%d %d\n", *pi, *pj);          // 10 20
*pi = 30;                          // spremeni tisto, na kar kaže pi – torej spremeni i
printf("%d %d\n ", i, *pj);           // 30 20
j = 40;
printf("%d %d\n ", *pi, *pj);         // 30 40
pi = pj;                              // odslej kaže pi na j, ne več na i
printf("%d %d\n ", *pi, *pj);         // 40 40
i = 50;
printf("%d %d\n ", *pi, *pj);         // 40 40
```

Reference

- Referenca ima lahko posebno vrednost null=0 (v C/C++) ali **nil** (v pascalu), kar pomeni, da ne kaže nikamor.
- Ta vrednost je večoblična (polimorfna) – je tipa void*

Reference

- Kos pomnilnika, na katerega bo kazal neka referenca, lahko zasedemo tudi dinamično (malloc vrne referenco na karkoli, kar moramo preoblikovati – *casting*):

```
#include <stdlib.h>                              // za malloc in free v C
typedef struct { int bar, baz; } Foo;
Foo *r = (Foo *) malloc(sizeof(Foo));            // v C
Foo *r = new Foo;                                // v C++
(*r). bar = 123; r->baz = 124;                   // operator ->
int *p = &r->bar;
printf("%d %d\n", *p, (*r).baz);                 // 123 124
*p = 125;
printf("%d\n", r->bar);                          // 125
free(r);                                         // v C
delete r;                                        // v C++
*p = 126;                                        // napaka!
```

Reference

- Imamo lahko opravka z referenco celotnega polja (naslov polja) in računanje z referencami:

```
Foo *t = (Foo *) malloc(10 * sizeof(Foo));        // v C
Foo *t = new Foo[10];                              // v C++
for (int i = 0; i < 10; i++) t[i].bar = 120 + i;
Foo *u = &t[3], *v = t + 5;                        // u kaže na t[3], v kaže na t[5]
printf("%d %d %d\n", u->bar, v->bar, v - u);       // 123 125 2
int *p = &u->bar;                                  // p kaže na t[3].bar
u += 2;                                            // u zdaj kaže na t[5]
*p = 105;
u->bar = 106;
printf("%d %d %d\n", t[3].bar, v->bar, u[2].bar); // 105 106 127
free(t);                                           // v C
delete[] t;                                        // v C++
```

Reference

- Parametri v podprograme in funkcije se prenašajo:
  - po vrednosti – naredi se kopija
  - po referfencei – prenes se naslov spremeljivke, ki jo prenašamo
- V C/C++ se polja prenašajo po referenci, kakor tudi predmeti (enako java)

Seznami

Seznam

- Seznam (*list*) je podatkovna struktura, ki hrani zaporedje elementov.
- Osnovni operaciji (lisp): car in cdr – glava in rep
- Da se ga implementirati na različne načine:
  - s tabelo (array)
  - z verigo (linked list)
  - veriga v tabeli

- Na seznamu lahko izvedemo sestavljene operacije, kot so:
  - dodajanje elementa
  - brisanje elementa
  - iskanje elementa (preverjanje, če je v seznamu)
  - sprehod skozi elemente (iteration)

- Od implementacije je odvisna tudi časovna zahtevnost teh operacij.

Seznam s tabelo

- Pripravimo si tabelo in hranimo elemente našega seznama v prvih nekaj celicah te tabele (implicitna podatkovna struktura):

```
typedef struct List_ {
  Element* a;    //  referenca tabele M celic
  int n, M;      //  seznam ima n elementov, a[0], a[1], …, a[n – 1]
} List;

void InitList(List& L, int M) {
  L.a = new Element[M];
  L.M = M; L.n = 0; }

void DoneList(List& L) {
  delete[] L.a; L.a = 0;
  L.n = 0; L.M = 0; }
```

Dodajanje na konec seznama s tabelo

- Dodajanje na koncu seznama je zelo poceni:

```
void AppendElement(List& L, Element e) {
  if (L.n == L.m) IncreaseArray(L);
  L.a[L.n] = e; L.n = L.n + 1; }
```

- Razen če je tabela že polna. Takrat je treba alocirati novo, večjo, in prenesti podatke vanjo, staro pa pobrisati:

```
void IncreaseArray(List& l) {
  L.M = 2 * L.M;
  Element *a = new Element[L.M];
  for (int i = 0; i < L.n; i++) a[i] = L.a[i];
  delete[] L.a; L.a = a; }
```

  - Novo tabelo smo naredili dvakrat tolikšno kot staro – podvajanje, lahko poljubna konstanta za povečanje.
  - Zato nastopajo potrebe po takšni realokaciji vse bolj poredko.
  - V povprečju je cena dodajanja enega elementa še vseeno $O(1)$.

Vstavljanje v seznam s tabelo

- Vstavljanje drugam v seznam je dražje, ker moramo ostale elemente premakniti naprej:

```
void InsertElement(List& L, int index, Element e) {
  assert(0 <= index && index <= L.n);
  if (L.n == L.M) IncreaseArray(L);
  for (int i = L.n - 1; i >= index; i++)
    L.a[i+1] = L.a[i];
  L.a[index] = e; L.n = L.n + 1; }
```

  - To je zdaj tem dražje, čim bolj pri začetku vrivamo element. V najslabšem primeru je lahko $O(n)$.

Brisanje elementa

- Stvar je podobna kot pri dodajanju: brisanje na koncu seznama je poceni, drugače pa moramo premikati ostale elemente:

```
void DeleteAtEnd(List& L) {
  assert(L.n > 0); L.n = L.n - 1; }

void Delete(List& L, int index) {
  assert(0 <= index && index < L.n);
  for (int i = index + 1; i < L.n; i++)
    L.a[i] = L.a[i+1];
  L.n = L.n - 1; }
```

- Lahko tudi označimo, da je mesto prazno ter ga kasneje ponovno uporabimo.
- Lažje je, če dovolimo, da se vrstni red ostalih elementov spremeni – prestavimo element s konca seznama v izpraznjeno celico:

```
void DeleteIgnoringOrder(List& L, int index) {
  assert(0 <= index && index < L.n);
  L.a[index] = L.a[n]; L.n = L.n - 1; }
```

- Če zbrišemo veliko elementov, se lahko zgodi, da postane tabela zelo prazna – če npr. pade *n* na *M*/4, lahko tabelo zamenjamo z manjšo, da ne bomo tratili pomnilnika.

Sprehajanje po seznamu

- Zelo enostavno je našteti vse elemente:

```
void Walk(List& L) {
  for (int i = 0; i < L.n; i++)
    DoSomethingWithElement(L.a[i]); }
```

- Iskanje elementa:

```
int FindElement(List& L, Element e) {
  for (int i = 0; i < L.n; i++)
    if (L.a[i] == e) return i;
  return -1; }
```

  – Žal traja to $O(n)$ časa.

Urejen seznam v tabeli

- Včasih želimo dodajati in brisati elemente tako, da je seznam ves čas urejen, torej da je $L.a[0] \leq L.a[1] \leq \ldots \leq L.a[n-1]$

```
void InsertSorted(List& L, Element e) {
  if (L.n == L.m) IncreaseArray(L);
  int i = L.n;
  while (i > 0 && L.a[i-1] > e) {
    L.a[i] = L.a[i - 1]; i = i - 1; }
  L.a[i] = e; L.n = L.n + 1; }
```

  - Brisanje bo enako kot prej pri `Delete` – ne smemo spremeniti vrstnega reda ostalih elementov.
  - Torej sta dodajanje in brisanje žal $O(n)$.

Razpolavljanje

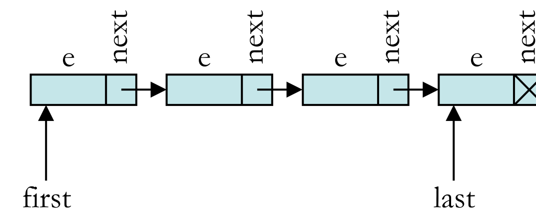
- Zakaj je dobro imeti urejen seznam v tabeli?
  - Elemente lahko naštejemo v naraščajočem vrstnem redu (Walk).
  - Iskanje je zdaj precej cenejše kot prej: bisekcija, $O(\log n)$.

```
int FindWithBisection(List& L, Element e) {
  if (L.n == 0 || e < L.a[0]) return 0;
  int left = 0, right = n;
  while (right - left > 1) {
    assert(L.a[left] <= e);
    assert(right == L.n || e < L.a[right]);
    int mid = (left + right) / 2;
    if (e < L.a[mid]) right = mid;
    else left = mid; }
  if (L.a[left] == e) return left; else return left + 1; }
```

  - Funkcija vrne indeks, na katerem se nahaja element *e*; če pa takega elementa ni, vrne indeks, kamor bi ga bilo treba vstaviti v seznam.
  - Invarianta v zanki je, da je $L.a[left] \leq e < L.a[right]$;
    pri $right = L.n$ si mislimo, da je tam $L.a[right] = \infty$.
  - Interval *left..right* se v vsaki iteraciji razpolovi,
    zato potrebujemo le $lg\ n$ iteracij.
  - Na koncu je $right = left + 1$, torej imamo $L.a[left] \leq e < L.a[left + 1]$
    - Torej je *e* bodisi na indeksu *left* bodisi ga ni v tabeli
      (in bi ga bilo treba vriniti na indeks *left* + 1)

Seznam kot veriga

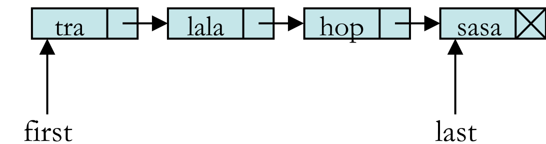
- Povezan seznam (*Linked list*) – eksplicitna podatkovna struktura:
  - Veriga ima po en člen za vsak element seznama.
  - Vsak člen alociramo posebej.
  - Vsak člen kaže na naslednjega v verigi.




```
typedef struct Node_ { // en člen verige
  Element e;           // element
  struct Node_* next;  // kazalec na naslednji člen
} Node;

typedef struct List_ {
  Node *first, *last;  // kažeta na prvi in zadnji člen verige
} List;

void InitList(List& L) {
  L.first = 0; L.last = 0; }

void DoneList(List& L) {
  while (L.first) {
    Node *next = L.first->next;
    delete L.first; L.first = next; }
  L.last = 0; }
```

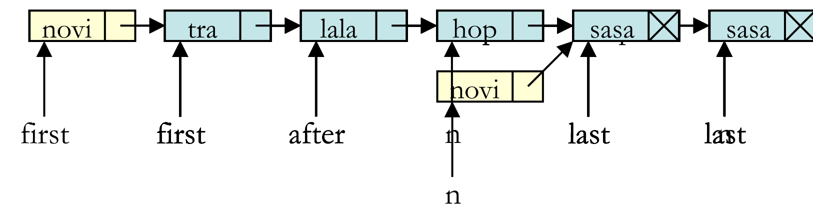
- Poraba pomnilnika je sorazmerna s številom elementov.
  - Ni potrate kot pri tabeli (ki ima lahko več celic kot elementov).
  - Je pa potrata zaradi kazalcev *next* (in morebitnih overheadov na kopici).

Dodajanje v verigo



- Ustvarimo nov člen, vanj vpišemo novi element in ga povežemo v verigo:

```
Node* NewNode(Element e, Node* next) {
  Node *n = new Node; n->e = e; n->next = next; return n; }

void AddToEnd(List& L, Element e) {
  Node *n = NewNode(e, 0);
  if (L.last) L.last->next = n;
  else L.first = n;
  L.last = n; }

void AddToStart(List& L, Element e) {
  L.first = NewNode(e, L.first);
  if (! L.last) L.last = L.first; }

void InsertAfter(List& L, Node *after, Element e) {
  Node *n = NewNode(e, after->next);
  after->next = n;
  if (L.last == after) L.last = n; }
```

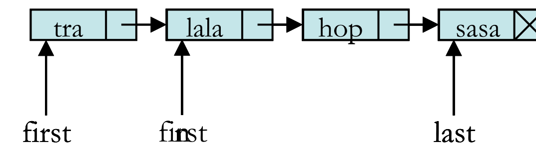


- Lepota: vse te operacije so *O*(1), neodvisno od dolžine seznama

Brisanje iz verige

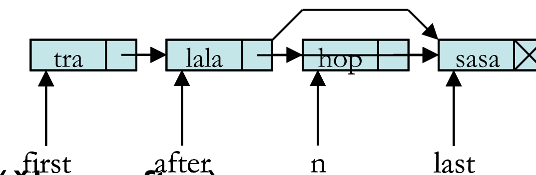
- Brisanje prvega elementa je enostavno:

```
void DeleteFirst(List& L) {
  if (! L.first) return;
  Node *n = L.first->next; delete L.first;
  if (L.last == L.first) L.last = n;
  L.first = n; }
```



- Brisanje drugod je malo bolj neugodno:
  - Na izbrisani člen kaže kazalec v njegovem predhodniku
  - Ta kazalec mora po novem kazati na naslednji člen (za izbrisanim)

```
void DeleteAfter(List& L, Node *after) { // zbriše člen za členom after
  Node *n = after->next; if (! n) return;
  after->next = n->next;
  if (L.last == n) L.last = after;
  delete n; }
```



  - Težava je v tem, da kazalca na predhodnika (člen *after*) nimamo nujno pri roki;
    če ga nimamo, moramo prečesati cel seznam

Brisanje iz verige

- Brisanje poljubnega elementa, če nimamo kazalca na predhodnika:

```
void Delete(List& L, Node *node) {
  if (node == L.first) { DeleteFirst(L); return; }
  Node *pred = L.first;
  while (pred && pred->next != node)
    pred = pred->next;
  assert(pred); // sicer node ni člen seznama L
  DeleteAfter(L, pred); }
```

- To pa zdaj traja v najslabšem primeru $O(n)$ časa
- Torej, če to operacijo res potrebujemo, je bolje imeti dvojno povezan seznam

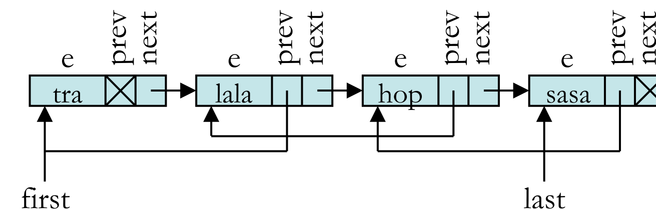
Dvojno povezan seznam

- (*Doubly linked list*) vsak člen kaže na naslednika in še na predhodnika
  - Zdaj moramo še bolj paziti, da pri prevezovanju kazalcev česa ne spregledamo

```
typedef struct Node_ {                // en člen verige
  Element e;                          // element
  struct Node_ *prev, *next;          // kazalec na prejšnji in na naslednji člen
} Node;
```

- Inicializacija in brisanje seznama se nič ne spremenita:

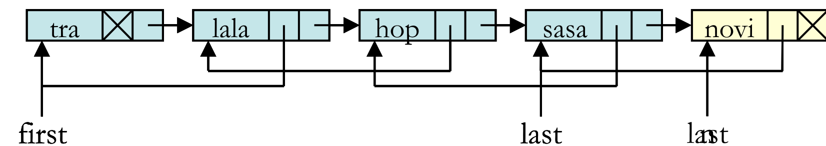
```
typedef struct List_ {
  Node *first, *last;  // kažeta na prvi in zadnji člen verige
} List;

void InitList(List& L) {
  L.first = 0; L.last = 0; }

void DoneList(List& L) {
  while (L.first) {
    Node *next = L.first->next;
    delete L.first; L.first = next; }
  L.last = 0; }
```



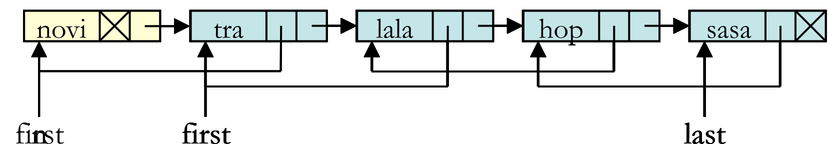
Dodajanje v dvojno povezan seznam

- Ustvarimo nov člen, vanj vpišemo novi element in ga povežemo v verigo:

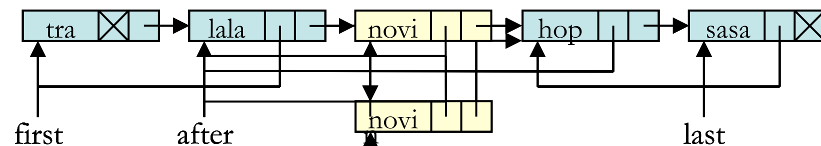
```
Node* NewNode(Element e, Node* prev, Node *next) {
  Node *n = new Node; n->e = e; n->prev = prev;
  n->next = next; return n; }

void AddToEnd(List& L, Element e) {
  Node *n = NewNode(e, L.last, 0);
  if (L.last) L.last->next = n;
  else L.first = n;
  L.last = n; }
```



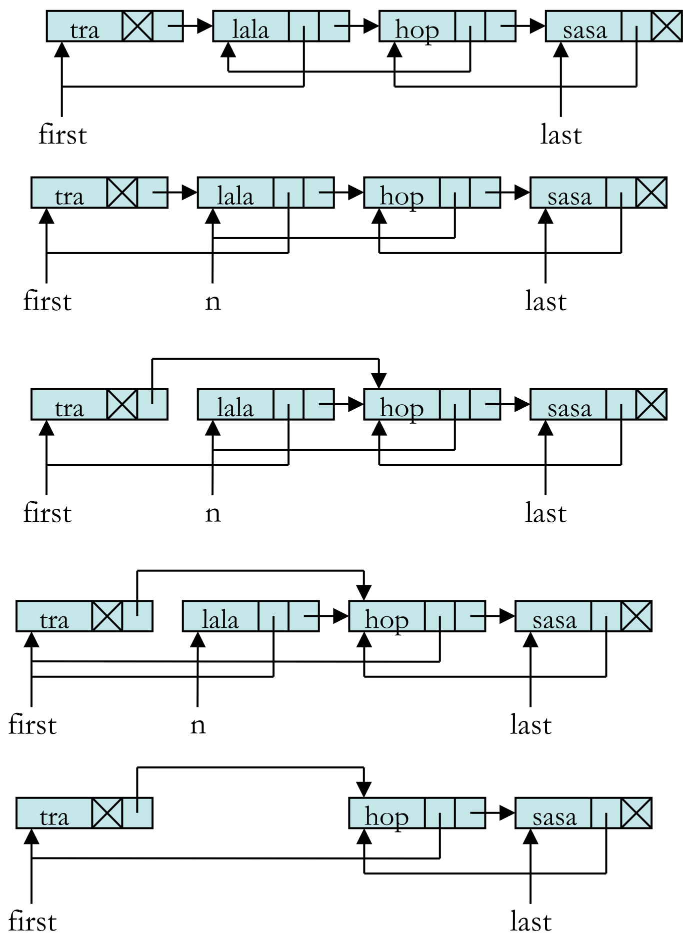

```
void AddToStart(List& L, Element e) {
  Node *n = NewNode(e, 0, L.first);
  if (L.first) L.first->prev = n;
  else L.last = n;
  L.first = n; }
```




```
void InsertAfter(List& L, Node *after, Element e) {
  Node *n = NewNode(e, after, after->next);
  after->next = n;
  if (n->next) n->next->prev = n;
  if (L.last == after) L.last = n; }
```



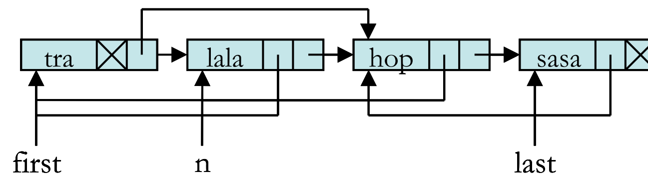
- Še vedno vse *O*(1), kot pri enojno povezani verigi


tra
lala
hop
sasa
first
last
tra
lala
hop
sasa
first
n
last
tra
lala
hop
sasa
first
n
last
tra
lala
hop
sasa
first
n
last
tra
hop
sasa
first
last

Brisanje iz dvojno povezanega seznama

- Zdaj lahko v *O*(1) pobrišemo katerikoli člen:

```
void Delete(List& L, Node *n) {
  if (n->prev) n->prev->next = n->next;
  else { assert(n == L.first); L.first = n->next; }
  if (n->next) n->next->prev = n->prev;
  else { assert(n == L.last); L.last = n->prev; }
  delete n; }
```



Iskanje po verigi

- Zelo enostavno je našteti vse elemente:

```
void Walk(List& L) {
  for (Node *n = L.first; n != null; n = n->next)
    DoSomethingWithElement(n->e); }
```

- Iskanje elementa:

```
Node* FindElement(List& L, Element e) {
  for (Node *n = L.first; n != null; n = n->next)
    if (n->e == e) return n;
  return 0; }
```

  - Žal traja to $O(n)$ časa.
  - Uporaba stražarja.
- Podobno kot pri seznamu v tabeli lahko tudi tu vzdržujemo urejen seznam.
  - ne moremo pa po njem iskati z bisekcijo, ker nimamo naključnega dostopa do elementov – to je ena od glavnih slabosti verige
  - če hočemo 123. element, moramo začeti na začetku in iti mimo prvih 122 elementov
  - različica: preskočni seznam (*skip list*)

Veriga v tabeli

- Doslej smo člene verige alocirali
  (`new, delete` / `malloc, free` / `New, Dispose` / ipd.)
- Včasih jih je koristno imeti v eni veliki tabeli
  - reference so sedaj indeksi v tabelo
  - neobstoječ element ima (npr.) indeks –1
  - s tabelo rokujemo kot smo opisali prej: če postane polna, jo podvojimo in če je preprazna, jo razpolovimo
- Zakaj je to dobro?
  - če vse skupaj shranimo na disk in kasneje spet naložimo, bodo indeksi v tabelo še vedno veljali – reference v pomnilnik pa ne bi imeli nobenega smisla več (prim. prenos predmetov v porazdeljenih okoljih)
  - če ima naš upravljalec s pomnilnikom veliko dela pri pridobivanju pomnilnika (načeloma jih sicer ne bi smel imeti)

Veriga v tabeli

- Primer:

```
typedef struct Node_ {             // ena celica tabele (prazna ali pa vsebuje en člen verige)
  Element e;                       // element
  int prev, next;                  // indeks prejšnjega in naslednjega člena
} Node;

typedef struct List_ {
  Node *a;
  int M;                 // a kaže na tabelo M celic
  int used;              // celice a[used], …, a[M – 1] so prazne
  int empty;             // indeks prve proste celice izmed celic a[0], …, a[used – 1]
  int first, last;       // indeksa prvega in zadnjega člena verige
} List;

void InitList(List& L, int M) {
  L.a = new Node[M]; L.M = M; L.used = 0;
  L.empty = -1; L.first = -1; L.last = -1; }

void DoneList(List& L) {
  delete[] L.a; L.a = 0; L.M = 0; L.used = 0;
  L.empty = -1; L.first = -1; L.last = -1; }
```

- Vse celice od *used* naprej so prazne
- Celice od 0 do *used* – 1 so lahko polne ali prazne
  - Prazne so povezane v enojno verigo, *empty* kaže na prvo od njih
  - Polne so povezane v dvojno verigo, *first* kaže na prvo od njih, *last* kaže na zadnjo od njih

	e	prev	next	
0	?	?	3	
1	?	?	7	
2	lala	4	6	
3	?	?	-1	
4	tra	-1	2	first = 4
5	?	?	9	
6	hop	2	8	
7	?	?	5	empty = 7
8	sasa	6	-1	last = 8
9	?	?	0	
10	?	?	?	used = 10
11	?	?	?	

n = 12

Dodajanje v verigo v tabeli

- Dodajanje elementa je enako kot pri verigi s kazalci, le alokacija novega člena je drugačna:

```
int AllocNode(List& L) {
  //  Če je v L.a[0], …, L.a[L.used – 1] še kakšna prazna celica,
  //  uporabimo prvo tako (in jo umaknimo iz verige praznih celic).
  int n = L.empty;
  if (n >= 0) { L.empty = L.a[n].next; return n; }
  //  Drugače pa bomo uporabili celico L.used (in povečali L.used za 1).
  if (L.used >= L.M) { //  Preselimo podatke v večjo tabelo.
    L.M = L.M * 2;
    Node *a = new Node[L.M];
    for (int i = 0; i < L.used; i++) a[i] = L.a[i];
    delete[] L.a; L.a = a; }
  return L.used++; }

int NewNode(List& L, Element e, int prev, int next) {
  int n = AllocNode(L); //  namesto new Node() kličemo AllocNode
  L.a[n].e = e; L.a[n].prev = prev; L.a[n].next = next;
  return n; }
```

Brisanje iz verige v tabeli

- Vse je enako kot prej, le namesto brisanja (`delete`) potrebujemo svoj podprogram:

```
void FreeNode(List& L, int n) {
  L.a[n].prev = -1;
  L.a[n].next = L.empty;
  L.empty = n; }
```

  - Pozor: to še ni brisanje elementa iz seznama, pač pa le sproščanje celice v tabeli.
  - Za brisanje potrebujemo podprogram, ki bo tudi prevezal kazalce (oz. po novem indekse) *prev*, *next*, *first*, *last*, enako kot prej pri navadni verigi

Primerjava seznama z tabelo in seznama z verigo

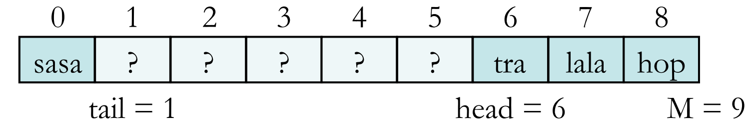
	Tabela	Veriga
Dodajanje na konec	$O(1)$*	vedno $O(1)$
Brisanje s konca	$O(1)$*	$O(n)$ [en.p.], $O(1)$ [dv.p.]
*povprečno; vendar $O(n)$, če je potrebna realokacija		
Dodajanje/brisanje na začetku	$O(n)$ 👎	$O(1)$
Vrivanje/brisanje vmes	$O(n)$ 👎	$O(1)$, če imamo kazalec na predhodnika ali je seznam dvojno povezan; sicer $O(n)$
Dostop do *k*-tega elementa	$O(1)$	$O(k)$ 👎
Dostop do prejšnjega elementa	$O(1)$	$O(n)$ [en.p.], O(1) [dv.p.]
Dostop do naslednjega elementa	$O(1)$	$O(1)$
Iskanje po urejenem seznamu	$O(\log n)$	$O(n)$
Pomnilniški overheadi	do n elementov	n ali $2n$ pointerjev + morebitni overheadi na kopici

Seznam je se uporablja kot oblika implementacije podatkovne strukture ***slovar***.
Uporaben, če so operacije omejene.

Vrsta

- Vrsta (*FIFO, queue*) je podatkovna struktura, pri kateri
  - dodajamo le na koncu (*tail*)
  - brišemo le na začetku (*head*)
  - a.k.a. FIFO (*first in, first out*)
- Operacije so:
  - enqueue – vstavi
  - dequeue – izloči
  - full – polna
  - empty – prazna

Vrsta



- Primerna implementacijo z verigo
- S tabelo pa ne, ker je brisanje prvega elementa drago
  - Lahko pa uporabimo krožno tabelo (*ring buffer*):

```
typedef struct Queue_ {
  Element *a; int M, head, tail; // ne vemo ali je povsem polna ali
                                 // povsem prazna
} Queue;

void InitQueue(Queue& Q, int M) {
  Q.a = new Element[M]; Q.M = M; Q.head = 0; Q.tail = 0; }

void Enqueue(Queue& Q, Element e) {
  if ((Q.head + 1) % Q.M == Q.tail) IncreaseArray(Q);
  Q.a[Q.tail] = e; Q.tail = (Q.tail + 1) % Q.M; }

Element Dequeue(Queue& Q) {
  assert(Q.head != Q.tail);
  Element e = Q.a[Q.head]; Q.head = (Q.head + 1) % Q.M; }

void IncreaseArray(Queue& Q) {
  Queue Q2; InitQueue(Q2, Q.M * 2);
  while (Q.head != Q.tail) Enqueue(Q2, Dequeue(Q));
  delete[] Q.a; Q = Q2; }
```

Sklad

- Pri skladu dodajamo in brišemo na istem koncu (vrh sklada, *top*)
  - LIFO = *last in, first out*
  - Dodajanju pogosto rečemo *push*, brisanju pa *pop*, obstaja še poizvedbe *top* – vrhnji, *full* in *empty*
- Kot nalašč za implementacijo s tabelo
- Lahko pa tudi z verigo (zadostuje enojno povezana – kazalec *first* kaže na vrh sklada)

- Mimogrede: (skoraj) vsi vaši programi že zdaj uporabljajo sklad, tudi če se tega ne zavedate
  - Tako je namreč organiziran del pomnilnika, kjer se hranijo lokalne spremenljivke in parametri pri klicih podprogramov

- Koristna vaja:
  - Implementiraj vrsto brez uporabe verige ali tabele, pač pa z dvema skladoma. Povprečna cena ene operacije na vrsti naj bo $O(1)$.

Vrsta z dvema koncema

- deque = double-ended queue
- Dodajamo in brišemo lahko na začetku in na koncu (ne pa vmes)
- Implementacija:
  - primerna je dvojno povezana veriga
  - primeren je tudi ring buffer, podobno kot za navadno vrsto

Razpršene tabele
(kot slovarji)

Razpršene tabele

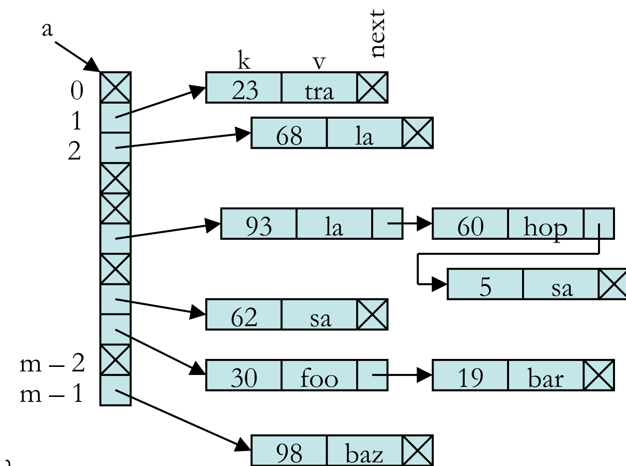
- Vsi poznamo navadne tabele (*arrays*):
  - Celice tabele so označene z indeksi
  - Indeksi so majhna nenegativna cela števila od 0 naprej: 0, 1, 2, …, n – 1 za tabelo z n elementi
  - Če poznamo indeks, lahko v času $O(1)$ zapišemo ali preberemo element na tem indeksu
- Včasih bi si želeli namesto z indeksom identificirati naše elemente z nečim drugim
  - Tej enolični oznaki elementa bomo rekli ključ (*key*)
  - Element je zdaj v bistvu par ⟨ključ, vrednost⟩
  - Vrednosti včasih pravijo tudi „spremljevalni podatki“/„satelitski podatki“
  - Ključ je mogoče niz znakov (npr. ime in priimek) ali pa veliko celo število (npr. EMŠO), ki ga ne moremo uporabiti kot indeks v tabelo

Razpršene tabele

- Ideja: izmislimo si neko funkcijo, ki preslikava ključe v indekse
  $h: K \to \{0, 1, \ldots, m-1\}$
  Temu bomo rekli razprševalna funkcija (hash function)
- Element s ključem *k* bomo v naši tabeli hranili v celici z indeksom *h*(*k*)
  - Vrednost *h*(*k*) je razprševalna koda (*hash code*) ključa *k*
  - Ups: *h* ponavadi ni injektivna – lahko se zgodi, da več različnih ključev dobi isti indeks – sovpadanje ali kolizija (*collision*)
  - Rešitev: vsaka celica tabele naj bo v resnici referenca na seznam, ki vsebuje vse ključe (in pripadajoče vrednosti), ki so se preslikali v indeks tiste celice – veriženje (*chaining*)

```
typedef struct Node_ {
  Key k; Value v;
  struct Node_ *next;
} Node;

typedef struct HashTable_ {
  Node **a;
  int m;
} HashTable;

void InitHashTable(HashTable& ht, int m) {
  ht.a = new Node*[m]; ht.m = m;
  for (int i = 0; i < m; i++) ht.a[i] = 0; }
```

Primer:
$m = 11$
Key = **int**
Value = **string**
$h(k) = k \% m$



  - Mimogrede: seznami obstajajo tui v drugih implementacijah

Dodajanje v razpršeno tabelo

- Preden dodamo nov element v razpršeno tabelo, preverimo, če element s takim ključem že obstaja
  - če že obstaja, spremenimo obstoječega, ne dodamo novega (mišljeno je, da ključ enolično identificira element)
  - druga možnost je multislovar

```
void PutValue(HashTable& ht, Key k, Value v) {
  int hc = HashFunction(k, ht.m); // hc = hash code
  for (Node *n = ht.a[hc]; n; n = n->next)
    if (n->k == k) { // tak ključ že obstaja
      n->v = v; return; }
  Node *n = new Node;
  n->k = k; n->v = v;
  n->next = ht.a[hc]; ht.a[hc] = n; }
```

Iskanje v razpršeni tabeli

- Izračunamo razprševalno kodo ključa, ki nas zanima, in pregledamo pripadajoči seznam
  - če ključ sploh je v tabeli, je v tem seznamu

```
bool GetValue(HashTable& ht, Key k, Value& v) {
  int hc = HashFunction(k, ht.m);
  for (Node *n = ht.a[hc]; n; n = n->next)
    if (n->k == k) { v = n->v; return true; }
  return false; }
```

Brisanje iz razpršene tabele

- Poiščemo ključ v seznamu in ga pobrišemo iz njega

```
bool DeleteKey (HashTable& ht, Key k) {
  int hc = HashFunction(k, ht.m);
  Node *prev = 0, *n = ht.a[hc];
  while (n && n->k != k) n = n->next;
  if (! n) return false;    // takega ključa ni
  if (prev) prev->next = n->next;
  else ht.a[hc] = n->next;
  delete n; return true; }
```

Časovna zahtevnost, *rehash*

- Videli smo: dodajanje, brisanje in iskanje potrebujejo $O(k)$ časa, če je k dolžina posamezne verige
- Ideja je, da naj bo m približno tolikšen, kolikor elementov hočemo hraniti v razpršeni tabeli
  - Tako bodo verige v povprečju zelo kratke, večinoma kar samo en element
  - Časovna zahtevnost je zato blizu $O(1)$
- Če se nam nabere veliko elementov in postanejo verige predolge, jih je smiselno preseliti v novo tabelo z večjim m (*rehash*) – prim. podvajanje

```
void Rehash(HashTable& ht, int mNew) {
  Node **aNew = new Node*[mNew];
  for (int i = 0; i < mNew; i++) aNew[i] = 0;
  for (int hcOld = 0; hcOld < ht.m; hcOld++)
    for (Node *n = ht.a[hcOld], *next; n; n = next) {
      next = n->next;
      int hcNew = HashFunction(n->k, mNew);
      n->next = aNew[hcNew]; aNew[hcNew] = n; }
  delete[] ht.a; ht.a = aNew; ht.m = mNew; }
```

Razprševalne funkcije

- Kako definirati primerno funkcijo $h(k)$?
  - Želeli bi, da bi čim bolj enakomerno pokrila množico možnih indeksov $\{0, 1, \ldots, m - 1\}$
  - Običajen pristop: $h(k) = h'(k) \bmod m$, pri čemer $h'(k)$ izračuna iz k neko veliko celo število
  - Za m radi vzamemo kakšno praštevilo, da bo ostanek odvisen od celega $h'(k)$ in ne morda le od zadnjih nekaj bitov; druga možnost je, da je $m=2^p$ in potem $h'()$ vključuje praštevilo
  - Primer za nize:

```
int HashFunction(const char *p, int m) {
  int hc = 0;
  while (*p) hc = ((hc << 8) | *p++) % m;
  return hc; }
```

  - Hash funkcije lahko tudi lepo kombiniramo med sabo: npr. če so naši ključi pari (k_1, k_2) in imamo že primerni hash funkciji $h_1(k_1)$ in $h_2(k_2)$, ju lahko skombiniramo npr. v
    $h(k_1, k_2) = (h(k_1) \cdot c + h(k_2)) \bmod m$
    za neko konstanto c – primerjaj zgoraj z nizi.

Razprševalne funkcije

- doslej smo sovpadanje reševela z veriženjem
- lahko rešujemo tako, da element, ki sovpade, poskusimo spraviti na neko drugo mesto v tabeli – odprto naslavljanje (*open addressing*)
- za iskanje naslednjega mesta potrebujemo novo funkcijo, ki izračuna indeks tega novega mesta: linearna, kvadratična, poljubna

- sodobne metode uporabljajo drugačen način iskanja praznega mesta – glej kukavičje razprševanje (*cuckoo hashing*)

- v vseh teh primerih:
  - v naprej vemo, kako velika bo razpršena tabela
  - ni dodatnih referenc
  - potrebno podvajanje in razpolavljanje

Urejanje

Urejanje

- Imamo zaporedje *n* elementov in bi jih radi preuredili tako, da bi na koncu veljalo $a[0] \le a[1] \le \ldots \le a[n-1]$
- Primer preprostega algoritma: urejanje z vstavljanjem (*insertion sort*)
  - Na začetku je *a*[0] sam zase urejeno zaporedje z enim elementom
  - Nato vsak naslednji element vstavimo v dosedanje urejeno zaporedje na tako mesto, da je novo zaporedje še vedno urejeno (vendar za en element daljše) – kot `InsertSorted`

```
void InsertionSort(Element *a, int n) {
  for (int i = 1; i < n; i++) {
    Element e = a[i]; int j = i - 1;
    while (j >= 0 && a[j] > e) { a[j+1] = a[j]; j--; }
    a[j+1] = e; }}
```

  - Časovna zahtevnost: v najslabšem primeru $O(n^2)$ – ko je bila vhodna tabela urejena ravno v obratnem vrstnem redu – kako pogosto se to zgodi?

Urejanje

- Še en preprost algoritem: urejanje z izbiranjem (*selection sort*)
  - Poiščemo največji element in ga postavimo na konec
  - Med preostalimi spet poiščemo največjega in ga postavimo na predzadnje mesto

```
void SelectionSort(Element *a, int n) {
  for (int i = n; i > 1; i--) {
    int m = 0;      //  m bo indeks največjega elementa izmed a[0], …, a[i – 1]
    for (int j = 1; j < i; j++)
      //  če je a[j] največji element doslej, si ga zapomnimo
      if (a[j] > a[m]) m = j;
    //  zamenjajmo največji element (m) z zadnjim (i – 1)
    Element e = a[m]; a[m] = a[i-1]; a[i-1] = e; }}
```

  - Porabimo vedno $O(n^2)$ časa (ne le v najslabšem primeru)

Kaj je to čas?

Deli in vladaj ter urejanje

- urejamo n elementov

  uredi (n elementov):

  1. ***razdeli na dva dela***
  2. ***uredi (prvi del); uredi (drugi del)***
  3. ***združi polovici***
  4. ***vrni urejene elemente***

- dve metodi:
  - vloži več dela v razdeljevanje in bo združevanje lažje; in
  - obratno
- razdelitev mora biti na čim bolj enaka dela; če sta velikosti različni za nek konstanten faktor => *O(n log n)*

Urejanje z zlivanjem - *Mergesort*

- Mergesort – več dela v združevanje:
  1. *O(1)*: razdelimo zaporedje na dva enaka dela ne glede na velikosti elementov
  2. *2T(n/2)*: uredimo prvi del posebej in drugi del posebej (z dvema rekurzivnima klicema)
  3. *O(n)*: urejeni zaporedji zlijemo skupaj
- časovna zahtevnost: *O(n log n)*
- prostorska zahtevnost: *O(n) + O(log n)*
- nelokalni dostop do polja

Hitro urejanje - *Quicksort*

- Quicksort – več dela v razdeljevanje:
  - naj bo *m* poljubna vrednost
  - razdelimo zaporedje na dva dela ("*partition*"):
    - v levem delu so sami taki elementi, ki so ≤ *m*
    - v desnem delu so sami taki elementi, ki so ≥ *m*
  - uredimo prvi del posebej in drugi del posebej (z dvema rekurzivnima klicema), bo s tem urejeno že tudi celotno zaporedje

  - bistvo: najti *m* tako, da bosta dela približno enako velika

Quicksort

- Delitev vhodne tabele na dva dela lahko elegantno izvedemo takole:
  - z enim števcem gremo z leve proti desni in elemente, ki so preveliki za v levi del, mečemo na desno
  - istočasno gremo z drugim števcem z desne proti levi in elemente, ki so premajhni za v desni del, mečemo na levo – jih preprosto zamenjujemo

```
void QuickSort(Element *a, int n) {
  if (n <= 1) return;
  Element m = f(a[nekaj od 0 do n-1]);
  int i = 0, j = n - 1;
  while (i <= j) {
    // Invarianta: a[0], …, a[i – 1] so ≤ m; med elementi a[i], …, a[n – 1] je vsaj eden, ki je ≥ m.
    // in:         a[j + 1], …, a[n – 1] so ≥ m; med elementi a[0], …, a[j] je vsaj eden, ki je ≤ m.
    while (a[i] < m) i++;
    while (a[j] > m) j--;
    if (i <= j) {
      Element e = a[i]; a[i] = a[j]; a[j] = e;
      i++; j--; }}
  QuickSort(a, j); QuickSort(a + i, n - i); }
```

Časovna zahtevnost quicksorta

- Če je partition razbil našo tabelo n elementov na dva dela, dolga po k in $n - k$ elementov, imamo:
  $T(n) = O(n)$ [za partition] + $T(k) + T(n - k)$ [za rekurzivna klica]
  - Če imamo srečo in sta oba dela približno enaka, $k \approx n/2$, se izkaže, da je $T(n) = O(n \log n)$
  - Če imamo smolo in je eden od obeh delov zelo kratek, drugi pa zelo dolg, npr. $k = n - 1$, se izkaže, da je $T(n) = O(n^2)$
  - ***V povprečju (če bi vzeli povprečje čez vse možne k-je) dobimo še vedno O(n log n) – torej moramo imeti za O(n²) res že hudo smolo***
  - Primer take smole: če za m vzamemo kar prvi element in nam nekdo podtakne že urejeno zaporedje
- Torej moramo pazljivo izbrati delilni element m:
  - Pogosto za m vzamejo srednji element: $m = a[n / 2]$
    - Tega bi se dalo tudi sesuti, a za to bi se moral nekdo posebej potruditi
  - Lahko vzamemo naključno izbran element: $m = a[rand() \% n]$
  - Lahko vzamemo tri naključne elemente in za m vzamemo srednjega med njimi (srednjega po velikosti)
  - Če je n majhen, se ne splača preveč komplicirati z izbiro m-ja
  - Pri res majhnih n (npr. n < 20) uporabimo raje insertion sort

Prostorska zahtevnost quicksorta

- Poraba dodatnega pomnilnika je odvisna od globine rekurzije
  - Npr. če se nam zgodi patološki primer, da tabelo vedno razdelimo na dva dela velikosti 1 in $n - 1$, bo šla rekurzija n nivojev globoko in porabili bomo $O(n)$ pomnilnika
  - Rešitev: drugi od obeh rekurzivnih klicev je repna rekurzija (*tail recursion*) – lahko ga spremenimo v iteracijo
    - Poskrbimo, da je drugi klic tisti, ki obdela večjega od obeh delov
    - Tako bo vsak rekurzivni klic gledal vsaj pol manjšo tabelo kot njegov klicatelj, zato bo šla rekurzija le $O(\log n)$ nivojev globoko
    - sodobni prevajalniki to naredijo že sami ne glede na našo odločitev

```
void QuickSort(Element *a, int n) {
  while (n < 20) {
    Element m = a[nekaj od 0 do n-1];
    int i = 0, j = n - 1;
    while (i <= j) {
      while (a[i] < m) i++;
      while (a[j] > m) j--;
      if (i <= j) {
        Element e = a[i]; a[i] = a[j]; a[j] = e;
        i++; j--; }}
    if (j < n - i) { QuickSort(a, j); a += i; n -= i; }
    else { QuickSort(a + i, n - i); n = j; }}
  InsertionSort(a, n); }
```

Z elementi lahko računamo

- Urejanje lahko bistveno učinkovitejše:
  - korensko urejanje
  - urejanje z vedri
- Obe metodi sta implicitni
- Časovna zahtevnost: $O(n * k)$, kjer je k odvisen od elementov, ki jih urejamo

Iskanje najmanjšega elementa

- Če imamo tabelo že urejeno, je to seveda trivialno
- Iskanje najmanjšega elementa:

```
int FindMin(Element *a, int n) {
  int m = 0;
  for (int i = 1; i < n; i++) if (a[i] < a[m]) m = i;
  return m; }
```

  - Porabili smo $n - 1$ primerjav.
  - Na enak način bi lahko z $n - 1$ primerjavami našli tudi največji element.
- Iskanje najmanjšega in največjega skupaj:

```
int FindMinMax(Element *a, int n, int& min, int& max) {
  if ((n % 2) == 1) min = 0, max = 0;
  else if (a[0] < a[1]) min = 0, max = 1;
                   else min = 1, max = 0;
  for (int i = (n % 2 ? 2 : 1); i < n; i += 2)
    if (a[i] < a[i + 1]) {
      if (a[i] < a[min]) min = i;
      if (a[i + 1] > a[max]) max = i + 1; }
    else {
      if (a[i + 1] < a[min]) min = i + 1;
      if (a[i] > a[max]) max = i; }}
```

  - Za iskanje obeh smo porabili le $3n/2$ primerjav namesto $2n - 2$.
- Tehnika pomnjenja - memoizacije

Iskanje *k*-tega najmanjšega elementa

- Če iščemo *k*-ti element le enkrat ali parkrat:

```
int KjeKtiNajmanjsi(Element *a, int n, int k) { // za 1 ≤ k ≤ n
  int m = -1;
  while (true) {
    int mm = -1, mc = 0;
    // V mm pripravimo indeks naslednjega elementa po velikosti (najmanjšega, ki je > a[m]).
    // V mc pripravimo število elementov s to vrednostjo.
    for (int i = 1; i < n; i++)
      if (m < 0 || a[i] > a[m])
        if (mm < 0 || a[i] < a[mm]) mm = i, mc = 1;
        else if (a[i] == a[mm]) mc += 1;
    if (k <= mc) return mm;
    k = k - mc; m = mm; }}
```

- Če je $k > n/2$, se najbrž bolj splača iskati od večjih proti manjšim
- Pojem amortizacijskega ali povprečnega časa: zaporedje večih operacij
- V vsakem primeru je to $O(nk)$, torej $O(n^2)$ – počasi

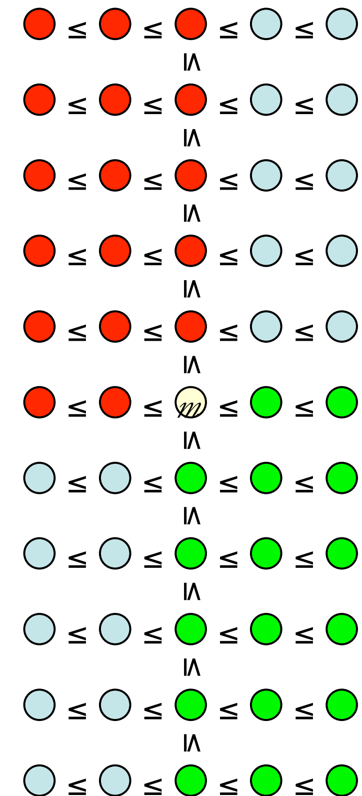
Iskanje *k*-tega najmanjšega elementa

- Hitro iskanje (prim. s hitrim urejanjem):
  – spet si izberimo nek element m in razdelimo tabelo na dva dela
    - vsi v prvem delu so $\leq m$
    - vsi v drugem delu so $\geq m$
    - torej je vsak iz prvega dela ≤ od vsakogar iz drugega dela
  – recimo, da je v levem delu r elementov, v desnem $n - r$, mi pa iščemo k-ti najmanjši element
    - če je $k \leq r$, vrnimo k-ti najmanjši element levega dela
    - sicer vrnimo $(k - r)$-ti najmanjši element desnega dela
    - (to naredimo z rekurzivnim klicem)
  – če imamo smolo (tako kot pri quicksortu), bo to $O(n^2)$, v povprečju pa bo le $O(n)$

Mediana median

- Razdelimo našo tabelo na $n/5$ skupin s po petimi elementi
- Za vsako skupino poiščimo mediano (srednji element po velikosti)
- Med temi $n/5$ medianami poiščimo njihovo mediano [rekurzivni klic] – recimo ji m
- Izkaže se:
  vsaj 30% elementov ● v naši tabeli je $\leq m$,
  vsaj 30% elementov ● v naši tabeli je $\geq m$
- Posledičnosmo za ceno rekurzivnega klica izločili 60% elementov ter lahko ponovimo iskanje rekurzivno na preostanku => O(n) primerjav za iskanje poljubnega elementa

- Zato je m odlično izhodišče za partition pri quicksortu ali pri iskanju k-tega najmanjšega elementa
  – Zagotavlja nam, da se patološki primeri s časovno zahtevnostjo $O(n^2)$ ne morejo zgoditi



Vrsta s prednostjo - kopica

Vrsta s prednostjo – kopica

- Vrsta s prednostjo (*priority queue*) vsebuje:
  - množico elementov
    - vsak vsebuje nek ključ in morebitne spremljevalne podatke
    - nad ključi mora biti definirana relacija urejenosti

  in učinkovito podpira naslednje operacije
  - dodajanje novega elementa
  - brisanje najmanjšega elementa
  - spreminjanje ključa pri že obstoječem elementu
  - (mogoče tudi) brisanje poljubnega elementa
  - (redkeje) združitev dveh kopic v eno (merge)

- Kako bi lahko podprli te operacije?
  - urejen seznam: dodajanje bo počasno
  - neurejen seznam: iskanje najmanjšega elementa + brisanje bo počasno
  - hash tabela: iskanje najmanjšega elementa bo počasno
  - binarno iskalno drevo: lahko, vendar je kopica preprostejša

- Ena najpogostejših implementacij vrste s prednostjo je kopica (*heap*)

Kopica

- Kopic je več vrst, v praksi je dovolj dobra že najpreprostejša – dvojiška kopica (*binary heap*)
  - binarno drevo (vsako vozlišče ima največ dva otroka)
  - popolnoma uravnoteženo (dolžina najdaljše in najkrajše veje se razlikujeta največ za 1)
  - globalno levo poravnano => možnost implicitne predstavitve
  - Vzdržujemo naslednjo lastnost: ključ v vsakem vozlišču je manjši ali enak kot v njegovih otrocih
    - posledica: v korenu je najmanjši ključ celega drevesa (kar bo koristno pri iskanju / brisanju najmanjšega ključa)

Kopica v tabeli

- Drevo ponavadi implementiramo s kazalci
  - vsako vozlišče je dinamično alocirano
  - vsebuje kazalce na očeta, otroke ipd.
- Kopica pa ima tako pravilno zgradbo, da jo lahko hranimo kar v tabeli:
  - oštevilčimo vozlišča po nivojih,
    na vsakem nivoju pa od leve proti desni (0 = koren)
  - vozlišče k ima otroka na $2k + 1$ in $2k + 2$ ter starša na $\lfloor (k - 1) / 2 \rfloor$; res za vsako popolno binarno drevo
  - torej ne potrebujemo referenc, da se premikamo gor in dol po kopici – so implicitne

Dodajanje v kopico

- Nov element dodajmo na konec kopice
- Če je manjši od očeta, ju zamenjajmo ("dvigovanje")
- Gornji korak ponavljamo, dokler je treba

```
typedef struct Heap_ {
  Element* a;   //  kaže na tabelo M celic
  int n, M;     //  kopica ima n elementov, a[0], a[1], …, a[n – 1]
} Heap;

void HeapAdd(Heap& H, Element e) {
  if (H.n == H.M) IncreaseHeap(H); //  povečajmo tabelo
  int idx = H.n++; //  novi element v mislih postavimo na konec
  while (idx > 0) {
    int parent = (idx - 1) / 2;
    if (H.a[parent] <= H.a[idx]) break;
    //  zamenjajmo starša in novi element
    H.a[idx] = H.a[parent]; idx = parent; }
  H.a[idx] = e; //  zdaj vemo, kam bomo res dali novi element, pa ga vpišimo
}
```

Brisanje korena

- Premaknimo v koren tisti element, ki je bil doslej zadnji v kopici
- Če je večji od kakšnega od otrok, ga zamenjajmo z manjšim od obeh otrok ("pogrezanje")
- Gornji korak ponavljamo, dokler je treba

```
void HeapDelRoot(Heap& H) {
  Element e = H.a[--H.n]; //  element s konca kopice
  int idx = 0;            //  element e v mislih postavimo v koren
  while (2 * idx + 1 < H.n) {
    int child = 2 * idx + 1;
    if (child + 1 < H.n)  //  poglejmo, kateri od otrok je manjši
      if (H.a[child + 1] < H.a[child]) child++;
    if (e <= H.a[child]) break;
    //  zamenjajmo e in manjšega od otrok
    H.a[idx] = H.a[child]; idx = child; }
  H.a[idx] = e; //  zdaj vemo, kam bomo res dali e, pa ga vpišimo
}
```

- Ta metoda je v povprečju slaba – obstaja boljša, ki sestoji iz pogrezanja praznine in dvigovanja elementa

Spreminjanje ključa

- Za začetek opozorimo, da poljubnega ključa ni enostavno najti v kopici
  - če ni manjši od korena, je mogoče v enem od poddreves, vendar ne vemo, v katerem
  - torej predpostavimo, da zna uporabnik kopice nekako drugače povedati, kateri element ga zanima
  - mogoče si mora med delom vzdrževati nekakšno “kazalo”, kjer za vsak element piše, kje v kopici se nahaja
- Ko spremenimo ključ našega elementa, je treba narediti nekaj od naslednjega dvojega:
  - če se je ključ zmanjšal, je treba element mogoče dvigniti
  - če se je ključ povečal, je treba element mogoče pogrezniti

Brisanje poljubnega elementa

- Kot prej – potrebno je vedeti, kateri element želimo brisati
- Sicer podobno kot brisanje korena:
  - na izpraznjeno mesto premaknemo element, ki je bil doslej na zadnjem mestu v kopici
  - če je ta element manjši od svojega novega očeta, ga dvigujemo, sicer pogrezamo

  - ali kako že z boljšo metodo?

Inicializacija kopice - *heapify*

- Recimo, da že imamo neko tabelo *n* elementov in bi jo radi organizirali v kopico
  - Preprosta rešitev:
    - začnimo s prazno kopico
    - dodajajmo vanje elemente enega za drugim
    - Slabost: $O(n \log n)$
  - Boljša rešitev:
    - mislimo si, da je tabela že kopica
    - pojdimo od spodnjih nivojev kopice proti višjim
    - vsak element pogreznimo, če je treba
    - na koncu imamo pravo kopico, časovna zahtevnost le $O(n)$
-

```
void Heapify(Heap& H) {
  for (int i = H.n / 2 - 1; i >= 0; i--) {
    int idx = i; Element e = H.a[idx]; // element, ki ga bomo pogrezali
    while (2 * idx + 1 < H.n) {
      int child = 2 * idx + 1;
      if (child + 1 < H.n)  // poglejmo, kateri od otrok je manjši
        if (H.a[child + 1] < H.a[child]) child++;
      if (e <= H.a[child]) break;
      // zamenjajmo e in manjšega od otrok
      H.a[idx] = H.a[child]; idx = child; }
    H.a[idx] = e; // zdaj vemo, kam bomo res dali e, pa ga vpišimo
  }
}
```

Uporaba vrste s prednostjo

- Urejanje:
  1. *O(n log n)*: vse elemente vstavimo v vrsto s prednostjo
  2. *O(n log n)*: iz vrste s prednostjo n krat poberemo najmanjši element

  – zagotovljeno *O*(*n* log *n*) tudi v najslabšem primeru

- Z uporabo kopice (*heap sort*):
  – tabelo, ki jo hočemo urediti, gledamo kot kopico
  – naredimo *heapify* (da bo res kopica)
  – pobrišimo največji element iz kopice (to je tisti iz korena) in ga prestavimo v zadnjo celico tabele (ki je zdaj prosta, ker je kopica za en element manjša)
  – prejšnji korak ponavljamo, dokler kopica ni prazna

  – v praksi je quicksort ponavadi malo hitrejši

Uporaba vrste s prednostjo

- Pogosto pride vrsta s prednostjo prav tam, kjer imamo opravka s ***požrešnimi algoritmi***:
  - Dijkstrov algoritem (najkrajše poti v grafu)
  - Primov algoritem (minimalno vpeto drevo v grafu)
  - Iskanje najmanjših *n* elementov v nekem zelo dolgem seznamu
  - Za predprocesiranje podatkov pri zunanjem urejanju (*merge sort*) – da nastanejo daljša že urejena podzaporedja (čete, *runs*)

Binarno iskalno drevo

Slovar

- Je podatkovna struktura, ki implementira abstraktno podatkovno strukturo slovar:
  - vzdrževanje množice ključev
  - dodajanje/brisanje ključa
  - iskanje po enakosti – prisotnost v množici
- Primerjaj z razpršilnimi tabelami
- Če dodamo urejenost nad elementi, lahko uporabimo drugačno podatkovno strukturo

Binarno iskalno drevo

- Če dodamo relacijo urejenosti, še operacije:
  - iskanje vseh ključev znotraj nekega intervala
    - naštejemo jih lahko v naraščajočem/padajočem vrstnem redu
  - iskanje prejšnjega/naslednjega ključa (glede na dani vrstni red)
  - iskanje najmanjšega/največjega ključa

  - primerjaj z vrsto s prednostjo

- Skoraj vse operacije (razen poizvedb, ki lahko vrnejo $O(k)$ ključev) podpira v času $O(\log n)$
  - razpršilna tabela podpira dodajanje, brisanje in iskanje po enakosti v $O(1)$ **pričakovanem**
  - ne podpira pa operacij, ki se nanašajo na vrstni red elementov

Binarno iskalno drevo

- Binarno iskalno drevo je rekurzivna podatkovna struktura:
  - kjer imamo koren in dve poddrevesi, ki sta lahko prazni (vsako vozlišče ima največ dva otroka)
  - vedno vzdržujemo naslednjo lastnost:
    - koren (pod)drevesa je večji od vseh ključev v levem poddrevesu in manjši od vseh ključev v desnem poddrevesu
  - primerjaj s kopico!

```
typedef struct Node_ {
  Key k;   // ključ
  SatelliteData d;   // spremljevalni podatki
  struct Node_ *left, *right;   // otroka
} Node;
```

Razširitev drevesa

- v korenu poddrevesa hranimo dodatne podatke o lastnostih drevesa:
  - število elementov, največji element, ...
- dodatno vzdržujemo lastnosti v korenih:
  - drevo.min = min (levi.min, desni.min, koren)
  - drevo.# = levi.# + desni.# + 1
  - ...

Iskanje po drevesu

- Začnimo pri korenu
  - če je drevo prazno => elementa ni
  - če je iskani ključ enak korenu, smo končali
  - če je manjši, moramo nadaljevati v levem poddrevesu
  - če je večji, pa v desnem poddrevesu

```
Node* TreeFind(Node *root, Key k) {
  Node *n = root;
  while (n) {
    if (k == n->k) return n;    // našli smo k
    if (k < n->k) n = n->left; // levo poddrevo
    else n = n->right; }        // desno poddrevo
  return n;
}
```

Dodajanje v drevo

- Začnemo enako kot pri iskanju
  - če vidimo, da ključ že obstaja v drevesu, končamo
  - sicer smo se ustavili, ker je drevo prazno in iz ključa naredimo novo drevo ter ga dodamo kot otroka

```
Node* TreeAdd(Node *root, Key k) {
  // dodajanje v prazno drevo bi morali obravnavati posebej (saj je trivialno)
  Node *n = root, **parent;
  while (n) {
    if (k == n->k) return n; // ključ k že obstaja v drevesu
    if (k < n->k) parent = &(n->left); // levo poddrevo
    else parent = &(n->right);          // desno poddrevo
    n = *parent; }
  n = new Node; // ustvarimo novo vozlišče
  n->k = k; n->left = 0; n->right = 0;
  *parent = n; // v očetu popravimo kazalec na novo vozlišče
  return n;
}
```

Brisanje iz drevesa

- Najprej poiščemo vozlišče, ki vsebuje iskani ključ *k*
- Če mu manjka eden od otrok, ga le zamenjamo s poddrevesom tega otroka in smo končali
- Sicer pobrišimo najmanjši element desnega poddrevesa (ali pa največji element levega poddrevesa) in tega vpišimo v vozlišče, iz katerega smo pred tem pobrisali *k*

Druge operacije

- Najmanjši element:
  - začnemo v korenu, sledimo povezavam v levo, dokler se da
- Predhodnik:
  - poiščimo ključ, katerega predhodnik nas zanima
  - recimo, da je v vozlišču *n*
  - če ima *n* levega otroka, vrnimo največji element levega poddrevesa
  - sicer se premaknimo v *n*-jevega starša in ponovimo prejšnji korak

Uravnoteženost drevesa

- Večina operacij nad drevesom ima časovno zahtevnost $O(h)$, pri čemer je h globina drevesa
  - to je načeloma $O(\log n)$, če je drevo kolikor toliko lepo uravnoteženo
  - lahko pa se drevo izrodi (npr. če začnemo s praznim drevesom in dodajamo elemente v naraščajočem vrstnem redu)

- Kaj storiti?

Uravnoteženost drevesa

- Elemente dodajajmo v naključnem vrstnem redu – **pričakovana** višina *O(log n)*
- Drevo sproti ravnotežimo, da ohranimo višino *c*log n*
  - AVL-drevo: poskrbimo, da se levo in desno poddrevo (v vsakem vozlišču) razlikujeta po globini največ za 1
  - rdeče-črno drevo: vsako vozlišče je rdeče ali črno, rdeče ne sme imeti rdečega starša, število črnih vozlišč vzdolž vsake veje drevesa je enako – dejansko posebna oblika B-dreves
  - B drevo: v vozlišču imamo med b/2 in b elementov in vsi listi drevesa so na isti globini
  - lomljena drevesa (*splay trees*), podobno kot pri move-to-front upravljanju z vrsto – pričakovana višina

Gradnja lepo uravnoteženega drevesa

- Recimo, da imamo *n* elementov, iz katerih bi radi zgradili drevo
- Uredimo jih naraščajoče in jih v tem vrstnem redu oštevilčimo: $a_0, \ldots, a_{n-1}$
  - $a_{n/2}$ dajmo v koren drevesa
  - iz $a_0, \ldots, a_{n/2-1}$ naredimo levo poddrevo (rekurzivni klic)
  - iz $a_{n/2+1}, \ldots, a_{n-1}$ naredimo desno poddrevo (rekurzivni klic)
- Tako dobimo drevo, ki bo karseda lepo uravnoteženo

Gradnja lepo uravnoteženega drevesa

- Recimo, da imamo *n* elementov, iz katerih bi radi zgradili drevo
- Naključno izberemo element iz tabele ter ga vstavimo v drevo, kar ponovimo za vse elemente
- Tako dobimo drevo, ki bo pričakovano karseda lepo uravnoteženo

in še in še ...

... toliko zanimivega je še ...

... a več jutri in še kdaj.

Hvala!