**STRUKTURE PODATAKA**

**I ALGORITMI 2**

ODGOVORI NA PITANJA ZA

USMENI DEO ISPITA

JANUAR 2021

Git: <https://github.com/NikolaVetnic/SPA2>

Sortiranje niza grubom silom i umetanjem.

01

**SORTIRANJE NIZOVA:** sortiranje po prirodnom uređenju koje postoji kada objekti implementiraju *Comparable* interfejs, sortiranje po eksternom kriterijumu koji je definisan komparatorom.

**SORTIRANJE GRUBOM SILOM:** dva elementa a[i] i a[j] su u inverziji ako je i < j i a[i] > a[j], pa se sortiranje grubom silom (*brute force*) svodi na poređenje svaka dva elementa ponaosob i njihovu razmenu ukoliko su u inverziji. Ukupan broj poređenja za n elemenata je n(n – 1)/2 pa je vremenska složenost O(n2) – u najgorem slučaju operacija je kvadratne složenosti veličine ulaza.

**SORTIRANJE UMETANJEM (INSERTION):** ideja je da se niz sastoji od sortiranog (sa leve strane) i nesortiranog dela, gde se prvi element iz nesortiranog ubacuje na odgovarajuće mesto u sortiranom (čime se sortirani uvećava za element), što se ponavlja dok se ne sortira čitav niz. Postupak: 1) početak je od prvog elementa koji predstavlja sortiran, a ostatak nesortiran deo; 2) neka je deo niza [0, i-1] sortiran – u svakom koraku se povećava za jedan, pa se i kreće u [1, arr.length – 1]; 3) prvi nesortirani element je arr[i]; 4) ukoliko je arr[i] > arr[i-1] ne radimo ništa budući da je on zapravo već sortiran, u suprotnom tražimo indeks j < i takav da je arr[j] ≤ arr[i] – arr[i] tada dolazi na poziciju j+1, elemente od arr[j+1] do arr[i-1] pomeramo za jedno mesto nadesno (što se vrši dok tražimo indeks j), pri čemu čuvamo referencu na arr[i] (jer će ga u pomeranju prebrisati arr[i-1]). Optimizacija: kod jezika koji ne podržavaju lenje računanje logičkih izraza potrebno je pre sortiranja pronaći minimalni element i staviti ga na početak.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA01_bruteforce_insertion>

Sortiranje niza izabiranjem i razmenom.

02

**SORTIRANJE IZABIRANJEM (SELECTION):** niz je sastavljen iz dva dela – sortiranog (sa desne strane) i nesortiranog, čime su svi elementi iz nesortiranog manji od svih iz sortiranog dela niza; traži se maksimalni element nesortiranog dela postavlja se na njegov kraj. Druga varijanta sortiranja deli niz na sortirani deo sa leve i nesortirani sa desne strane; traži se minimum desne strane i postavlja na početak nesortiranog dela niza.

**STABILNA I NESTABILNA SORTIRANJA:** postupak sortiranja je stabilan ako za svako i i j, takvo da je a[i] = a[j] u polaznom nizu, element a[i] u sortiranom nizu bude pre a[j] – relativni redosled identičnih elemenata biva očuvan kod stabilnog sortiranja. Po *default*-u sortiranje umetanjem je stabilno a izabiranjem nije (iako se može napraviti da bude). Stabilna sortiranja dozvoljavaju sortiranje po dva kriterijuma, i to prvo po sekundarnom, a onda po primarnom – sortiranje po sekundarnom ostaje očuvano zahvaljujući stabilnosti sortiranja.

**SORTIRANJE RAZMENOM (BUBBLE):** zasniva se na ideji da će, ako svaka dva susedna elementa koja su u inverziji zamene mesta, maksimalni element „isplivati“ na kraj niza (kraj nesortiranog dela varira od n – 1 kada je nesortirani deo čitav niz, do 1 kada nesortirani deo niza čine dva elementa); inverzni postupak ima analogni efekat na minimalni element (početak nesortiranog dela varira od 0 do n – 2). Optimizacija: sortiranje se može unaprediti tako što se sortiranje prekine posle iteracije u kojoj nije bilo razmene elemenata.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA02_selection_bubble>

Sortiranje niza *Shell sort*-om.

03

**SHELL SORT:** (*Donald Shell*, 1959) donekle liči na *insertion sort*; osnovna ideja – 1) ukupan broj parova u inverziji se može posmatrati kao mera neuređenosti niza, 2) kod elementarnih sortiranja pri svakom poređenju susednih elemenata ukupan broj inverzija se smanjuje za jedan, 3) razmena nesusednih elemenata u nizu može smanjiti više od jedne inverzije. Niz A dužine n se deli na podnizove A0, A1… Ak-1 za neko k na sledeći način: A0 = (0, k, 2k…), A1 = (1, k+1, 2k+1…) … Ak-1 = (k-1, 2k-1…) – drugim rečima, podniz Aj sadrži indekse koji pri deljenju sa k daju ostatak j; podnizovi se sortiraju umetanjem; postupak se ponavlja za vrednosti k koje se smanjuju od neke početne vrednosti do 1 sa nekim korakom; kada je k = 1 dešava se sortiranje umetanjem nad celim nizom.

Kako varirati k: 1) sam *Shell* predložio je vrednosti , , … , , … , 1, gde je n veličina niza (*Shell sort* se značajno ubrzava kada k varira tako da bude stepen broja 2 uvećan ili umanjen za 1), 2) (*Tokuda*, 1992) sortiranje daje najbolje performanse kada se k sukcesivno smanjuje 2.25 puta; 3) (*Ciura*, 2001) sortiranje daje najbolje performanse kada k uzima redom vrednosti { 701, 301, 132, 57, 23, 10, 4, 1 }.

Demo: <https://www.youtube.com/watch?v=SHcPqUe2GZM>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA03_shell>

Sortiranje niza *Comb sort*-om.

04

**COMB SORT:** donekle liči na *bubble sort*; zasniva se na razmeni nesusednih elemenata u nizu, predstavlja unapređenje *Bubble sort*-a (sortiranje razmenom); kod sortiranja razmenom veći elementi „brzo prilaze“ kraju niza, dok manji „sporo prilaze“ početku niza – osnovna ideja je eliminisati „efekat kornjača“ („sporih“ elemenata) razmenom nesusednih elemenata u nizu.

**Algoritam**: 1) za svaki element proveravamo da li je u inverziji sa elementom udaljenim k pozicja udesno i ako jeste razmenimo ih, 2) ponavljamo *Comb* prolaz smanjujući k, 3) niz je sortiran kada je k = 1 ili nije bilo razmena elemenata[[1]](#footnote-1). Kako varirati k – eksperimentalno je utvrđeno da *Comb sort* daje najbolje performanse kada se k sukcesivno smanjuje 1.3 puta, ali se preskoče k = 9 i k = 10 i uzme se k = 11.

Demo: <https://www.youtube.com/watch?v=n51GFZHXlYY>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA04_comb>

Sortiranje niza *Heap sort*-om.

05

**HEAP SORT:** unapređenje sortiranja izabiranjem linearitamske složenosti (O(n logn)).

**Algoritam**: 1) neka je A niz dužine n, 2) transformišemo A tako da ime strukturu *heap*-a (maksimum je tada prvi u nizu), 3) razmenimo maksimum sa elementom na kraju niza, 4) povratimo strukturu *heap*-a na podnizu A[0 .. n-1] i maksimum zamenimo sa elementom na poz. n – 2, 5) ponavljamo postupak do podniza A[0 .. 1] kada maksimum zamenimo sa susednim elementom.

***Heap* osobina**: niz ima *heap* osobinu ukoliko je svaki element veći ili jednak svojim sinovima, gde sinovi roditelja indeksa p imaju indekse 2p + 1 i 2p + 2, a roditelj sina indeksa s ima indeks (s – 1)/2 (celobrojno deljenje); kod niza dužine n elementi sa idx u opsegu [0, (n – 1)/2] su očevi. Počevši od poslednjeg ka prvom ocu uspostavljamo *heap* strukturu tako što: 1) proverimo da li je otac manji od većeg sina, 2) ako jeste, zamenimo oca sa većim sinom, 3) ponavljamo **1)** i **2)** dokle god otac ne postane veći od sinova.

**public** **static** **<T..** **>** **void** **sort(T[] arr)**: uzima se idx poslednjeg elementa i poslednjeg oca i u while petlji se uspostavlja *heap*; int end dobija vrednost kraja niza i ulazi kao kontrola while petlje tokom koje koren i poslednji element menjaju mesta, end se dekrementira i ponovo se uspostavlja *heap*.

**private** **static** **<T..** **>** **void** **makeHeap(T[] arr, int start, int end)**: int parentIndex uzima start, uvodi se flag heapRestored; ulaz u while petlju sa uslovom !heapRestored, računa se maxSonIndex za parentIndex i proverava se da li je == -1 – ako da petlja se prekida, inače sledi provera da li je element sa idx oca manji od onog sa idx sina – ako da zamenjuju mesta i parentIndex uzima vrednost sina (dakle, parentIndex ide ka većoj vrednosti), inače se petlja prekida.

**private** **static** **<T..** **>** **int** **getMaxSon(T[] arr, int parentIndex, int end)**: računaju se dva moguća idx za datog oca; provera da li je prvi sin van opsega niza, ako nije maxSonIndex uzima njegovu vrednost; provera da li je drugi sin van opsega niza, ako nije provera da li je njegov element veći od elementa sa maxSonIndex, ako jeste maxSonIndex uzima njegovu vrednost; vraća se maxSonIndex (čija *default* vrednost je -1).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA05_heap>

Sortiranje niza *Quick sort*-om (opis i implementacija Lomutove šeme particionisanja i bar još jedne po izboru).

06

**QUICK SORT:** (*Tony Hoare*, 1959) jedan od najčešće korišćenih u praksi, u proseku linearitamske vremenske složenosti (O(n logn)), zasniva se na *divide and conquer* principu. Ideja: 1) odaberemo jedan element koji nazovemo pivot, 2) preuredimo niz tako da je oblika (LE pivot QE), gde su LE elementi manji od ili jednaki pivotu, a QE veći ili jednaki pivotu (postupak particionisanja), 3) sortiramo LE i QE *quick sort*-om (najčešće realizovano rekurzivno). Particionisanje niza: selekcija pivota i transformacija u formu (LE pivot QE), za čega postoji više šema (*Hoarova* – pivot je A[l], *Lomutoova* – pivot je A[h], i ona gde je pivot na sredini niza). Opšti oblik za prve dve šeme:

public static <T extends Comparable<T>> void sort(T[] arr) {

sort(arr, 0, arr.length – 1); }

private static <T extends Comparable<T>> void sort(T[] arr, int l, int h) {

int j = partition(arr, l, h);

sort(arr, l, j – 1);

sort(arr, j + 1, h); } }

***Hoarova* šema**: za pivot se uzima prvi element niza (idx l), kreiraju se idx i (na l+1 poziciji) i j (na kraju niza); idx i se kreće nadesno preskačući sve elemente strogo manje od pivota dok ne dođe do elementa x koji ne ispunjava taj uslov – analogno idx j preskače strogo veće od pivota dok ne dođe do elementa y, kada se dešava zamena x i y, nakon čega se kretanje idx i, pa j ponavlja; kada se idx i i j mimoiđu (j < i) dolazi do zamene pivota i el. sa idx j jer on pripada delu koji je manji od pivota.

Demo: <https://www.youtube.com/watch?v=cnzIChso3cc>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA06_quick_hoare>

***Lomutova* šema**: za pivot se uzima poslednji element niza (idx h) i sekvencijalno se prolazi kroz čitav niz od idx l do h-1 i tom prilikom se niz deli na dva dela – sa elementima manjim ili jednakim pivotu i elementima strogo većim od pivota; svaki sledeći element (idx i) može biti: 1) manji ili jednak pivotu – tada se korišćenjem posebnog idx ltePivot on menja sa prvim elementom dela niza većeg od pivota čime se deo manji ili jednak pivotu uvećava za jedan, i 2) veći od pivota – tada se ne radi ništa; finalni korak je razmena pivota sa idx ltePivot.

T pivot = arr[h];

int ltePivot = l - 1;

for (int i = l; i < h; i++) {

if (arr[i].compareTo(pivot) <= 0) {

++ltePivot;

T tmp = arr[i];

arr[i] = arr[ltePivot];

arr[ltePivot] = tmp;

} }

int placeForPivot = ltePivot + 1;

T tmp = arr[placeForPivot];

arr[placeForPivot] = arr[h];

arr[h] = tmp;

return placeForPivot;

Demo: <https://www.youtube.com/watch?v=86WSheyr8cM>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA06_quick_lomuto>

**Sredina niza**: idx i kreće od l i preskače strogo manje, idx j od h i preskače strogo veće, pivot je na (i+j)/2; kada se zaustave idx i i j dolazi do zamene; postupak se ponavlja dok se idx i i j ne mimoiđu, s tim da ovde pivot ne ostaje na mestu već završi u delu niza suprotnom od idx koji je pre stigao do njega (desnom ako je prvi i, npr.); slede dva rekurzivna poziva za opsege [l,j] i [i,h]. Nema provere izlaska iz opsega niza jer se pivot ponaša kao graničnik za onaj idx koji prvi stigne do njega.

Demo: <https://www.youtube.com/watch?v=Hoixgm4-P4M>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p03_advanced_sorts>

Sortiranje liste umetanjem.

07

**SORTIRANJE UMETANJEM:** svaki postupak za sortiranje niza može se adaptirati za sortiranje listi, ali neki postupci su prirodniji listama (npr. baš sortiranje umetanjem je prirodno listama jer se element trivijalno umeće na drugo mesto u listi, dok se kod niza moraju pomerati elementi).

Ako je lista prazna ili jednoelementna ona je sortirana. Neka je levi deo sortiran a desni nesortiran, tada: lastSorted je pokazivač na poslednji element u sortiranom (deklaracija i inicijalizacija VAN while petlje), a firstUnsorted na prvi u nesortiranom delu (pa je firstUnsorted = lastSorted.next); razlikuju se tri slučaja: 1) firstUnsorted >= lastSorted – nema umetanja, sortirani deo se trivijalno povećava (lastSorted = firstUnsorted); 2) firstUnsorted < root – firstUnsorted se umeće pre root-a; 3) inače – firstUnsorted treba umetnuti iza lastLeq (lastLeq pokazuje na poslednji element između root i firstUnsorted koji je manji ili jednak od firstUnsorted, što se lako pronalazi pretraživanjem liste od korena ka kraju).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA07_list_insertion>

Sortiranje liste *Merge* sortiranjem.

08

**MERGE SORT:** zasniva se na *divide and conquer* principu, linearitamske složenosti (O(n logn)). Osnovna ideja: 1) podelimo listu L u dve balansirane liste L1 i L2 (dužine L1 i L2 se razlikuju najviše za 1), 2) sortiramo L1 i L2 *merge sort*-om, 3) spojimo dve sortirane liste L1 i L2 u finalnu sortiranu listu L. Sortiranje se tako sastoji od dva potproblema: 1) podela liste u dve balansirane liste, i 2) spajanje sortiranih listi u (jednu) sortiranu listu.

**Podela liste**: neka start pokazuje na koren originalne liste koja ima bar dva elementa, l1 i l2 pokazuju na korene L1 i L2, l1End i l2End pokazuju na poslednje elemente L1 i L2; podela počinje inicijalizacijom pokazivača:

l1 = start; l1End = l1; l2 = start.next; l2End = l2;

Tako l1 i l1End pokazuju na koren, a l2 i l2End na drugi element originalne liste; nakon inicijalizacije ažuriraju se samo l1End i l2End (elementi iz originalne liste se dodaju na kraj L1 ili L2); elementi se dodaju u L1 i L2 naizmenično, i to tako što se dodaju na kraj liste, nakon čega se povrati esencijalno svojstvo current pokazivača.

Na kraju, voditi računa o tome da element koji ide u L1 može biti poslednji (kod liste sa neparnim brojem elemenata) pa se pre dodavanja u L2 mora raditi provera na null; takođe nakon deljenja potrebno je „uzemljiti“ L1 i L2: l1End.next = null; l2End.next = null;

**Spajanje sortiranih listi**: neka root pokazuje na koren liste koja se dobija spajanjem dve sortirane liste L1 i L2, last pokazuje na poslednji element u spojenoj listi, l1 i l2 na prvi element koji nije dodat u spojenu iz L1 i L2 (na početku l1 i l2 pokazuju na korene L1 i L2). Spajanje se realizuje u tri koraka:

1) određivanje korena spojene liste – ako je prvi element L1 manji od prvog elementa L2 tada (analogno u obrnutom slučaju):

root = l1; l1 = l1.next; (povraćaj esencijalne osobine l1)

last = root; (prvi el. spoljne liste je na početku i njen poslednji)

2) dodavanje elemenata iz L1 i L2 u spojenu listu – razlikujemo slučaj kada je l1.info < l2.info i tada dodajemo l1 na kraj spojene liste (analogno u obrnutom slučaju) – ova operacija se ponavlja dokle god postoje elementi u obe liste:

last.next = l1;

last = l1; (povraćaj esencijalne osobine last)

l1 = l1.next; (povraćaj esencijalne osobine l1)

3) kalemljenje ostatka kada se jedna od listi „isprazni“ – vrši se jednostavnom proverom i prevezivanjem pokazivača:

last.next = l1 == null ? l2 : l1;

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA08_list_merge>

Sortiranje liste *Quick* sortiranjem.

09

**QUICK SORT:** bazira se na ideji pivota i podele liste na dva dela spram pivota; neka je L = (G|R) lista koja se sortira, gde je G pivot (za koji selektujemo element koji se najbrže i najlakše dobavlja – koren) a R ostatak liste koju delimo na dva dela: M (manji od G) i V (veći od G); sortiramo M i V i dobijamo M' i V', pa je tada sortirana lista L' = (M'|G|V'). Formiranje lista M i V se realizuje obilaskom liste G. Budući da sortirana lista L' ima oblik L' = (M'|G|V') lista M' može biti prazna lista – tada je koren L' liste čvor G a ne prvi el. iz M'; ako pak M' nije prazna, tada G treba nakalemiti na kraj M' (šetnja do kraja liste M'); kalemljenje V' na G je trivijalno.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA09_list_quick>

ADT prioritetna lista, operacije *insert*, *max* i *isEmpty*.

10

**PRIORITETNA LISTA:** kolekcija objekata u kojoj svaki element ima prioritet obrade (ideja uvođenja prioriteta tako da se podaci sa većim prioritetom ranije procesiraju), pri čemu se element sa najvećim prioritetom procesira prvi; elementi nisu nužno sortirani po prioritetu, bitno je samo da se u svakom trenutku zna koji ima najveći prioritet.

**Prioritet**: 1) eksplicitni – svakom elementu se pridružuje ceo/realan broj kojim je određen prioritet, i 2) implicitni – prioritet određen prirodnim uređenjem (objekti klase koja implements Comparable, MAX-PQ i MIN-PQ) ili proizvoljnim komparatorom.

Tip podataka ADT-a određuje *skup vrednosti i operacija* nad varijablama tog tipa – način realizacije ADT-a *sakriven od korisnika tipa* (korisnik ne mora da zna detalje implementacije kako bi koristio tip). Kod ADT prioritetne liste **vrednost tipa** je kolekcija uporedivih elemenata (može biti struktuirana na više fundamentalno različitih načina), a **operacije** su: 1) dodavanje novog elementa, 2) dobavljanje elementa sa najvećim prioritetom, 3) brisanje elementa sa najvećim prioritetom, 4) provera da li je lista prazna, 5) veličina liste.

**Specifikacija ADT-a**: apstraktni deo ADT-a; zaglavlja public metoda kojima se definišu operacije ADT-a, to je ono što korisnik treba da zna da bi koristio tip; različite implementacije ADT-a dele istu specifikaciju. U Java-i se ADT može specifirati interfejsima:

public interface PriorityQueue<T extends Comparable<T>> {

void insert(T element); T max();

T delMax(); boolean isEmpty();

int size(); }

**Reprezentacija prioritetne liste**: izvodi se nizom koji ima *heap* osobinu, što obezbeđuje efikasne *insert/delete* operacije logaritamske složenosti (O(logn)); ovakav niz se može predstaviti u obliku binarnog stabla, što je struktura koja se sastoji od čvorova pri čemu svaki čvor ima pokazivače na levo i desno podstablo. Terminologija: 1) svaki čvor ima najviše dva sina; 2) koren nije nikome sin, a listovi nemaju sinove; 3) otac čvora B je A ako je B sin od A.

**BINARNO STABLO SA *HEAP* OSOBINOM:** zadovoljava sledeće uslove: 1) stablo je kompletno (svi nivoi osim možda poslednjeg su potpuno popunjeni, a poslednji ako nije potpun je popunjen redom sleva nadesno; 2) za svaki čvor važi da je veći od svojih sinova (odnosi se na oba podstabla, koren je maksimum). Ovakva stabla jednostavno se predstavljaju nizom, gde su sinovi sa indeksima 2p + 1 i 2p + 2, a roditelj čvora sa indeksom s je (s – 1) / 2 (celobrojno deljenje).

**void insert(T element)**: 1) dodaje se novi element na kraj (proširivog) niza; 2) popravlja se niz tako da se ponovo uspostavi *heap* osobina preko indeksa dodatog sina.

***void restoreHeapProperty(int sonIndex)***: računa se indeks oca i ako je novi element veći od njega razmenjuju se, što se ponavlja sve dok je zadovoljen uslov razmene ili dok se ne stigne do korenskog čvora (dakle, sonIndex ide naniže).

**T max()**: proverava se da li je lista prazna i ako nije vraća se koren liste (idx = 0).

**boolean isEmpty()**: proverava da li je veličina liste jednaka 0.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA10_11_priority_queue>

ADT prioritetna lista, *delMax* i *size*.

11

**T delMax()**: 1) poslednji element niza zameni mesta sa korenom; 2) povrati se *heap* struktura metodom void restoreHeapProperty() – bez argumenata!

***void restoreHeapProperty()***: metoda identična istoimenoj kod *heap sort*-a; kretanjem od korena ka kraju niza pronalazi se idx većeg od dva sina metodom int getMaxSon(int parentIndex); ukoliko takav ne postoji trenutni element je list i *heap* osobina je uspostavljena, ukoliko postoji upoređuju se otac i (veći) sin – ukoliko je otac veći *heap* osobina je uspostavljena, ukoliko nije otac i (veći) sin zamene mesta, sin postaje otac i postupak se ponavlja (dakle parentIndex ide od 0 naviše).

***int getMaxSon(int parentIndex)***: računaju se idx oba (potencijalna) sina, izlazna promenljiva se inicijalizuje na -1, proverava se da li postoji element sa idx prvog, a zatim i drugog sina – ako ne postoji izlazna promenljiva dobija vrednost prvog sina, a ako postoji upoređuju se sinovi i izlazna dobija vrednost većeg.

**int size()**: vraća se veličina niza.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA10_11_priority_queue>

ADT skup realizovan otvorenih hešovanjem.

12

***HASH* TABELA**: niz u kom se pozicija elementa određuje na osnovu njega samog pomoću *hash* f-je koja izračunava poziciju: h : Object → [0 .. M-1], gde je M veličina hash tabele; *hash* f-ja mora da bude brzo i lako izračunljiva. *Hash* tabelama realizuju se dva bitna ADT-a: skupovi i mape, budući da je pretraživanje znatno brže u odnosu na realizacije nizovima i listama.

**ADT SKUP**: kolekcija objekata bez duplikata. Operacije: 1) dodavanje elementa, 2) brisanje elementa, 3) pretraživanje – da li je element u skupu, 4) unija, presek, razlika (često su dovoljne samo prve tri, uz napomenu da i dodavanje i brisanje zahtevaju pretraživanje).

**ADT MAPA**: mapa, iliti asocijativni niz / tabela simbola / rečnik, je kolekcija objekata oblika (ključ, vrednost) pri čemu svaki objekat ima jedinstven ključ (nema duplikata ključeva). Operacije: 1) dodavanje (k, v) para u mapu – provera da li već postoji k u mapi i nastavljamo ako ne; 2) brisanje (k, v) para iz mape; 3) pretraživanje (*lookup*) – dobavljanje vrednosti za dati ključ (null ukoliko ne sadrži k), najbitnija operacija; 4) modifikovanje v za dati k.

***HASH* KOD**: *hash* f-ja vraća vrednost – *hash* kod – jedinstvenu za objekat, i ona mora biti konzistentna (k1 = k2 → hash(k1) = hash(k2)); metod hashCode() definisan u klasi Object koristi se da se kreira *hash* f-ja u opštem slučaju. *Hash* f-ja vraća broj u opsegu [0 .. M-1] za veličinu tabele M (veza između *hash*-a i *hash* koda):

public int hash(Object o, int hashTableSize) {

return Math.abs(o.hashCode() % hashTableSize); }

Redefinišući metod hashCode() u našoj klasi dobijamo *hash* kod za objekte te klase, a oni se mogu proslediti u gornju metodu hash() kako bismo dobili vrednosti *hash* f-je za naše objekte. **Kolizije**: dva objekta mogu imati istu vrednost *hash* f-je i to se naziva kolizijom (verovatnoća kolizije je neintuitivan (nelinearan) fenomen, veća je nego što intuicija sugeriše)[[2]](#footnote-2). **Princip uniformnosti**: neka je M veličina *hash* tabele i О skup objekata gde je |O|>>M, tada *hash* f-ja particioniše О u familiju skupova Hk, 0≤k<М, gde Hk sadrži objekte sa istom vrednošću *hash* f-je; dobra *hash* f-ja: |H0| ≈|H1| ≈ ... ≈|HM-1| ≈ |O|/M

**OTVORENO HEŠOVANJE (ZATVORENO ADRESIRANJE)**: jedan od mehanizama razrešenja kolizija, *hash* tabela je niz JPL, liste (lanci kolizija) sadrže objekte sa istom vrednošću *hash* f-je; za veličinu *hash* tabele bira se neki veliki prost broj. Demo: <https://tinyurl.com/g4g-hashSet>

**Vrednosti**: da bi se napravio skup objekata klase A koristeći OHashSet klasa A mora da redefiniše sledeće metode klase Object: int hashCode(), boolean equals(Object o), String toString().

**private static class Node { … }**: *wrapper* klasa za realizaciju lanaca kolizije, sadrži Object value i Node next pokazivače.

**OHashSet()**, **OHashSet(size)**, **hash(T o)**: konstruktori i hash f-ja (hashCode() % table.length).

**Node[] searchCollisionChain(T element, int hashValue)**: krećući se od korena lanca kolizije proverava za svaki element u lancu da li je jednak traženom i ako jeste vraća niz od dva pokazivača, i to na traženi element i onaj koji mu prethodi.

**boolean member(T element)**: pretraga lanca kolizije za *hash* vrednost elementa.

**boolean insert(T element)**: pretraga lanca kolizije za *hash* vrednost elementa, ukoliko postoji vraća false, ukoliko ne postoji element se stavlja u Node i prevezuje na početak lanca.

**boolean remove(T element)**: pretraga lanca kolizije za *hash* vrednost elementa, ukoliko ne postoji vraća false, ukoliko postoji pita se da li je to koren lanca i ako jeste izvršava table[hashValue]=table[hashValue].next, a ako nije n[1].next=n[0].next (n je nizovni rezultat pretrage lanca kolizije).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qA12_open_hash_set>

ADT skup realizovan zatvorenim hešovanjem, operacija *insert*.

13

**ZATVORENO HEŠOVANJE**: jedan od mehanizama razrešenja kolizija, *hash* tabela je niz gde je pozicija objekta o u nizu hash(o); svaka ćelija u *hash* tabeli je: EMPTY, OCCUPIED (zauzeta, sadrži element) ili DELETED (nekad sadržala element koji je sada obrisan); dodavanje novog elementa n: 1) računa se hash vrednost za n; 2) ako je table[hash] EMPTY tada table[hash]=n i prelazi u OCCUPIED; 3) ukoliko je *pozicija zauzeta* određuje se *nova pozicija po nekoj strategiji*: a) linearno probavanje (hash(o)+1, hash(o)+2...) – *uzrokuje klastere* (usporava pretraživanje), b) kvadratno probavanje (hash(o)+1, hash(o)+4, hash(o)+9...) – traži se *prva dostupna ćelija sa statusom EMPTY*, usled čega je lanac kolizija uslovljen *strategijom probavanja* i zato može sadržati elemente različitih vrednosti *hash* f-je.

**Operacije**: 1) **pretraživanje** tabele, odnosno lanca kolizija za dati element, 2) **dodavanje** novog elementa u tabelu – na kraj lanca kolizija ili u prvu DELETED ćeliju, 3) **brisanje** iz tabele – lenjo brisanje (status ćelije se menja u DELETED).

**Lanac kolizija**: *hash* tabela je niz dužine M (veliki prost broj) pa zato lanac kolizija ima maksimalnu dužinu – kod *kvadratnog probavanjа* se za maksimalnu vrednost uzima (M–1)/2; prvih (M–1)/2 kvadratnih probavanja su uvek *probavanja različitih lokacija u tabeli*, posle (M–1)/2 neuspešnih probavanja proširuje se tabela (takođe slučaj kada je opterećenje tabele veće od 70%, odnosno kada je više od 70% ćelija OCCUPIED).

**public class CHashSet<T> implements Set<T> { … }**: od polja postoji niz objekata, niz statusa (enum E/O/D) sa idx koji odgovaraju idx objekata i broj elemenata.

**OHashSet()**, **OHashSet(size)**, **hash(T o)**, **reset()**: konstruktori, hash f-ja i anuliranje polja.

**int searchCollisionChain(T element, int hashValue)**: hashValue postaje currentPosition, prati se brojač i uzima se maxChainLength = (table.length – 1) / 2; prolazak petljom kroz lanac i vraćanje currentPosition ukoliko se pronađe element, u suprotnom se vraća -1.

**boolean insert(T element)**: uzimaju se *hash* elementa i maksimalna dužina lanca (M–1)/2, prati se brojač i, kraj lanca i prva dostupna pozicija; kretanje kroz petlju dok se ne dođe do kraja lanca ili dok brojač ne dostigne maksimalnu dužinu – ako je pozicija zauzeta ekvivalentnim elementom vraća se false, inače se proverava da li je već pronađena prva dostupna pozicija (ako nije – uzima se trenutna) i da li se došlo do kraja lanca kolizije. Nakon petlje ukoliko ne postoji dostupna pozicija ili je opterećenje veće od 70% tabela se proširuje i element se dodaje metodom add(element, hashValue), a ukoliko postoji se dodaje.

**void add(T element, int hashValue)**: kretanje kroz lanac na sličan način kao sa insert(T element) metodom, ali ovaj put bez ijedne provere osim da li je pronađena dostupna pozicija.

**void expand()**: veličina tabele je najmanji prost broj koji je barem dva puta veći nego što je trenutna veličina tabele; postupak: 1) napravi se kopija stare tabele i statusa; 2) realociramo tabelu i postavimo statuse svih ćelija na EMPTY; 3) iteriramo kroz staru tabelu i ćelije u statusu OCCUPIED dodajemo u novu tabelu; 4) staru tabelu i statuse postavimo na null vrednosti.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT skup realizovan zatvorenim hešovanjem, operacije *member* i *remove*.

14

**int searchCollisionChain(T element, int hashValue)**: uzima se *hash* elementa koji se traži kao trenutna pozicija, prate se brojač i maksimalna dužina lanca (M–1)/2; kretanje kroz while dok je brojač manji od maks. dužine i dok je status trenutne pozicije != EMPTY – ako je status OCCUPIED proverava se sadržaj elementa i ako su jednaki vraća se trenutna pozicija, u suprotnom se ažuriraju brojač i trenutna pozicija; na kraju se vraća trenutna pozicija ili -1 ukoliko element nije pronađen.

**boolean member(T element)**: prosto se iskoristi se pretraga lanca kolizije.

**boolean remove(T element)**: koristi se pretraga lanca kolizije za nalaženje pozicije traženog elementa; ukoliko je element pronađen status sa njegovom pozicijom se postavi na Status.DELETED a broj elemenata se smanji.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT mapa realizovana otvorenim hešovanjem, operacije *insert* i *delete*.

15

**ADT MAPA, OTVORENO HEŠOVANJE**: čvorovi u tabeli sada sadrže parove (ključ, vrednost), a *hash* f-ja se primenjuje samo na ključeve (drugim rečima, lanac kolizija sadrži samo elemente čiji ključevi imaju istu vrednost *hash* f-je).

**private static class Node { … }**: *wrapper* klasa za realizaciju lanaca kolizije, sadrži Object key, Object value i Node next pokazivače.

**OHashMap()**, **OHashMap(size)**, **hash(T o)**: konstruktori i hash f-ja (hashCode() % table.length).

**Node[] searchCollisionChain(K key, int hashValue)**: lista kolizija se pretražuje tražeći čvor sa odgovarajućim ključem; ukoliko takav čvor postoji u tabeli vraća se niz od dve reference, i to na traženi čvor, kao i na onaj koji mu prethodi.

**boolean insert(K key, V value)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ pronađe vraća se false, inače se kreira novi element i postavlja na početak liste.

**boolean delete(K key)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ ne pronađe vraća se false, inače se pitamo da li je ključ na početku liste – ako jeste, prevezuje se sam početak liste, a ako nije, prevezujemo rezultat pretrage liste njemu prethodnim elementom.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

ADT mapa realizovana otvorenim hešovanjem, operacije *get* i *modify*.

16

**V get(K key)**: u praksi se često dešava da postoje elementi koji se dobavljaju češće u odnosu na prosek (*hot* i *cold data*) – primenjuje se LRU strategija (*least recently used* – poslednji traženi element pomeri se na početak lanca kolizija). Pretražuje se lista kolizija za *hash* vrednost ključa i ukoliko se ne pronađe vraća se null; inače, koristeći oba pokazivača koje vraća pretraga traženi element se prevezuje na početak liste, nakon čega se kao rezultat vraća početak liste kolizija.

**boolean modify(K key, V value)**: pretražuje se lista kolizija za *hash* vrednost ključa; ukoliko se takav ključ ne pronađe vraća se false, inače postavljamo vrednost na argument sa kojim je pozvan metod i vraćamo true.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p05_hash_tables>

Pretraživanje sa vraćanjem – opšti opis postupka i modifikacije.

01

**PRETRAŽIVANJE SA VRAĆANJEM**: algoritamski postupak – ideja ili šablon primenjiv na širi spektar algoritamskih problema, *backtracking* je algoritamski postupak primenjiv na široku klasu problema poput 1) kombinatorne enumeracije – pronalaženje *svih konfiguracija* sistema koje *zadovoljavaju određeni kriterijum* (primeri: svi podskupovi skupa pozitivnih brojeva čiji je zbir jednak broju T, svi rasporedi 8 kraljica na šahovskoj tabli tako da se kraljice međusobno ne napadaju) i 2) kombinatorne optimizacije – pronalaženje konfiguracije sistema koja zadovoljava određeni *optimizacioni kriterijum* (primer: problem trgovačkog putnika – naći najkraću turu na mapi gradova).

**Konfiguracija** kombinatornog sistema može se predstaviti vektorom/nizom C = (c0, c1, ..., ct) čije elemente nazivamo komponentama (primer: kraljice na tabli – vektor od 8 komponenti koje su tada koordinate polja na kojima su kraljice). **Konfiguracija podskupa** skupa S – logički vektor C od k komponenti, gde je k kardinalnost skupa S; C[i] = true / false → S[i] jeste / nije u podskupu. **Parcijalna konfiguracija kombinatornog sistema** – vektor Ck, k < t, kod kojeg su prvih k komponenti poznate (određeni), dok su ostale nepoznate (neodređene).

***BACKTRACKING***: problem – pronaći sve konfiguracije C koje zadovoljavaju logički predikat P; osnovna ideja: 1) na osnovu P odredimo predikat Q koji je primenjiv na parcijalne konfiguracije sistema kojim definišemo da li je neka parcijalna konfiguracija Ck proširiva (primeri: kod zbira Q = postoji Ck+1 u kojoj je zbir elemenata ≤ T, kod kraljica Q = postoji Ck+1 u kojoj se kraljice ne napadaju), 2) parcijalne konfiguracije koje ne zadovoljavaju Q zovemo konfiguracijama neuspeha, 3) a one koje zadovoljavaju Q zovemo potencijalnim konfiguracijama; *backtracking* je dakle sistematično (iscrpno, sveobuhvatno) proširivanje potencijalnih konfiguracija.

**STABLO *BACKTRACK*-A**: proširivanje potencijalnih konfiguracija se može predstaviti *backtracking* stablom, gde je koren stabla prazna konfiguracija, unutrašnji čvorovi stabla potencijalne konfiguracije, listovi ciljne (koje zadovoljavaju P) ili konfiguracije neuspeha; ako je čvor A sin čvora B – konfiguracija A se dobija proširivanjem potencijalne konfiguracije B.

***BACKTRACKING*, DALJE**: *backtracking –* „obilazak“ stabla u dubinu: 1) ne konstruiše se stablo *backtrack*-a već postoji jedna konfiguracija koja se menja **proširivanjem** i **sažimanjem**; 2) dinamika promene te parcijalne konfiguracije korespondira obilasku stabla *backtrack*-a u dubinu; 3) parcijalna konfiguracija se proširuje u ciljnu (konfiguracija koja zadovoljava P) – imamo rešenje; 4) ako se parcijalna ne može proširiti tada se **sažima i nastavlja se sa proširenjima sažete parcijalne konfiguracije** (sažimanje – izbacivanje poslednje komponente iz parcijalne konfiguracije); 5) proširenje parcijalne konfiguracije ne uspeva iz dva razloga – ili su već ispitana sva proširenja, ili proširenje vodi u konfiguraciju neuspeha; 6) **sažimanje prve parcijalne konfiguracije** – **kraj *backtracking*-a**.

**Proširivanje**: mora se uvesti red prilikom proširivanja trenutne parcijalne konfiguracije kako se ne bi proširivala na neku konfiguraciju koja je već bila ispitana; za trenutnu parcijalnu konfiguraciju dužine k-1 formira se Dk: lista dopustivih elemenata za k-tu komponentu (kako imamo t komponenti to će biti t listi dopustivih elemenata); trenutna parcijalna konfiguracija Ck-1 se proširuje tako što: 1) f = prvi element liste Dk, 2) obriše se f iz Dk, 3) formira se Ck proširenjem Ck-1 sa f. *Backtrack* nerekurzivno:

Vector C; // konfiguracija sistema

void backtracking() {

D[] = new D[C.length]; // liste dopustivih elemenata

int k = 0; // trenutna komp. (dužina parc. konfig.)

compute D[0]; // odredi dopustive el. za prvu komp.

while (k >= 0) {

while (!D[k].empty) {

f = remove first element in D[k];

C[k] = f;

if (k == C.length – 1)

printConfiguration(); // imamo rešenje

else {

k++; // idemo na sledeću komponentu

compute D[k]; } } // odredi dopustive el. za sledeću komp.

k--; } } // vraćamo se nazad

// Određivanje dopustivih el. na osnovu predikata Q primenjenog na trenutnu parcijalnu konfiguraciju.

**Algoritam *backtracking*-a**: počinje se od vektora konfiguracije sistema C; računa se lista dopustivih elemenata za k = 0, gde je k dužina parcijalne konfiguracije, odnosno njena trenutna komponenta; ulazi se u while petlju koja traje sve dok se parcijalna konfiguracija ne sažme nazad do početne, odnosno sve dok je k ≥ 0; unutar petlje se uzima prvi dopustiv element i njime se fiksira k-ta komponenta parcijalne konfiguracije; sledi provera da li je dužina parcijalne konfiguracije jednaka dužini vektora C – ako jeste znači da imamo rešenje, a ako nije prelazi se na sledeću komponentu i računa se lista dopustivih za nju; na kraju se, unutar while petlje, k smanjuje za jedan, odnosno radi se sažimanje ukoliko nije preostalo dopustivih elemenata za trenutnu parcijalnu konfiguraciju. *Backtracking* rekurzivno:

Vector C; // trenutna konfiguracija sistema

void backtracking() {

backtracking(0); }

backtracking (int k) { // k – trenutna komponenta

compute D; // određivanje liste dopustivih elemenata za k-tu komp.

while (!D.empty()) {

f = remove first element in D;

C[k] = f;

if (k == C.length – 1)

printConfiguration();

else

backtracking (k+1); } }

Računanje dopustivih elemenata je moguće optimizovati: ukoliko za komponente C znamo da su u nekom intervalu [min, max] tada umesto liste dopustivih elemenata čuvamo samo jedan, i to najmanji dopustivi element, dok se sledeći određuje na osnovu prethodnog (primer: kraljice – vrste/kolone su u intervalu [0 .. 7]:

Vector C;

backtracking () {

Vector D;

int k = 0;

D[0] = first available element >= min

while (k >= 0) {

while (D[k] <= max) { // dok D[k] nije uzeo max dopustivu vrednost…

C[k] = D[k] // C[k] se fiksira sa D[k]

D[k] = first available element > D[k] or max + 1 (no available elements)

if (k == C.length – 1) printConfiguration();

else {

k++;

D[k] = first available element >= min or max + 1 (no available elements) } }

k--; } }

Rešenje problema kraljica pretraživanjem sa vraćanjem (objasniti i implementirati bar jednu verziju algoritma po izboru).

02

**PROBLEM N KRALJICA**: rasporediti N kraljica na šahovsku tablu dimenzija N \* N tako da nema kraljica koje se međusobno napadaju (a napadaju se ako: 1) x1 = x2, 2) y1 = y2, 3) |x1 – x2| = |y1 – y2|) – koliko ima takvih rasporeda i koji su (rešenje po vrstama)? Rešenje se predstavlja nizom int[] table, gde su kraljice na pozicijama (i, table[i]), i < n.

**VERZIJA 1 – LISTE DOPUSTIVIH POLJA**: unutrašnja klasa AvailableCell *wrapper* za položaj u vrsti sa pokazivačem na sledeći (lista dopustivih polja), polja klase su veličina table n, brojač rešenja slCounter, raspored kraljica int[] table i niz listi dopustivih polja; konstruktor prima veličinu n i inicira table i availableCells, printSolution() uveličava broj rešenja i štampa trenutno; find() metod: 1) formiranje liste raspoloživih polja za prvu vrstu[[3]](#footnote-3) i postavljanje currentRow na 0; 2) pokreću se dve while petlje, gde se prva prekida kada se konfiguracija sažme na manje od nule (odnosno kada smo isprobali sve konfiguracije), a druga kada više nema raspoloživih polja; 3) table[currentRow] uzima vrednost prvog raspoloživog polja, koren raspoloživih polja prelazi na sledeći element[[4]](#footnote-4); 4) proverava da li postoji rešenje sa (currentRow == n – 1) – ako da štampa se i beleži, ako ne izvrši se currentRow++ i traži nova lista raspoloživih polja; findAvailableCells(int currentRow) metod: 1) postavlja listu za trenutni currentRow na null; 2) kretanje kroz dve for petlje, gde prva ide od poslednje ka prvoj koloni (jer su tako popunjene dosadašnje kraljice), a druga od prve kolone ka poslednjoj i proverava da li su polja u koliziji sa već postavljenim kraljicama po uslovu #1 (table[i] == j – provera po vertikali) i uslovu #3 (|i – currentRow| == |table[i] – j| - provera po dijagonali, horizontala pa vertikala) – uslov #2 se ne proverava jer su vertikale već „ograđene“ postavkom for petlji; 3) ako je polje prošlo proveru dodaje se na početak liste (polja su tako sortirana neopadajuće).

**VERZIJA 2 – ČUVANJE JEDNOG DOPUSTIVOG ELEMENTA**: umesto availableCells sada postoji int[] firstAvailableCell (sadrži prvo dostupno polje za vrstu i); find() metod: 1) formiranje liste raspoloživih polja za prvu vrstu prosto se postavi firstAvailableCell[0] na 0; 2) druga while petlja se sada prekida kada je firstAvailableCell[currentRow] < n; 3) table[currentRow] uzima vrednost prvog raspoloživog polja, currentRow se povećava i pokreće se updateFirstAvailableCell(currentRow); 4) u else grani takođe postoji ažuriranje dopustivog polja; updateFirstAvailableCell(int currentRow) metod: 1) kretanje kroz dve for petlje, gde prva ide od prvog dostupnog polja za trenutni red do kraja reda, a druga od prvog reda do trenutnog; 2) druga petlja proverava koliziju sa već postavljenim kraljicama; 3) ukoliko provera prođe, uzima se trenutna vrednost brojača prve for petlje za prvo raspoloživo polje; 4) ukoliko nijedna ćelija nije dostupna, vrednost prvog raspoloživog polja se stavlja na n (izlaz iz traženja rešenja).

**VERZIJA 3 – REKURZIVNO**: prvo dostupno polje sada se računa tokom traženja rešenja; find() metod: 1) izlaz je currentRow == n; 2) prolazak kroz čitav red od 0 do n brojačem j, provera da li je polje (currentRow, j) dostupno i ako jeste uzima se i traži se rešenje za currentRow + 1; availableCell(int currentRow, int currentColumn) metod: 1) prolazak for petljom od 0 do currentRow; 2) provera kolizije za vrednost table[i].

**public void find()**:poziva se privatna metoda find(0).

**private void find(currRow)**: ako je currRow == n štampa se rešenje, inače se for petljom prolazi kroz kolone i proverava se da li je dostupna pozicija [currRow][j] – ako jeste table[currRow] dobija vrednost j i rekurzija ide dalje.

**private bool availableCell(currRow,** **currCol)**: prolazak for petljom kroz sve redove i provera na koliziju sa već postavljenim kraljicama (po vertikali i dijagonali).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB02_nqueens/>NQueensRec.java

Rešenje problema ruksaka pretraživanjem sa vraćanjem (objasniti i implementirati bar jednu verziju algoritma po izboru).

03

**PROBLEMI KOMBINATORNE OPTIMIZACIJE**: probleme kombinatorne optimizacije često je moguće formulisati na opšti način: 1) dat je skup S od n objekata, 2) svaki objekat ima neku vrednost (težinu, cenu, profit) predstavljenu f-jom f: S → Integer, 3) f-ja f: S → Integer je definisana i na partitivnom skupu skupa S i aditivna je: f(T) = zbir f(x) za svako x iz T, gde je T ⊂ S, 4) potrebno je naći podskup skupa S koji zadovoljava određena ograničenja definisana logičkim predikatom P tako da se f maksimizuje (ili minimizuje).

**PROBLEM RUKSAKA**: dat je skup S od n predmeta pri čemu svaki ima težinu (*weight*) i cenu (*profit*); dat je ranac u koji se smeštaju predmeti čija težina ne prevazilazi maksimalnu težinu knapsackWeight; problem: napuniti ranac predmetima iz S tako da se maksimizuje profit, što se u opštem slučaju ne dešava gramzivim punjenjem. Opšta formulacija:

Set greedy(Set S) {

Sort S in non-increasing order by f or some function derived from f

Set solution = empty set

for (int i = 0; i < S.length; i++) {

if (solution U {S[i]} satisfies P) {

solution = solution U {S[i]} } } }

return solution }

Ideja gramzivih algoritama je da lokalno optimalne odluke vode globalno optimalnom rešenju, kao i da se jednom doneta odluka više ne može opozvati; gramzive strategije za punjenje ranca: 1) sortiranje po profitu, ili 2) po težini (najmanja ka najvećoj), ili 3) po „gustini profita“ (*profit / weight*); konkretan primer dat na slajdovima sa predavanja.

**BACKTRACK REŠENJE**: tražimo sve podskupove T skupa predmeta S koji zadovoljavaju kriterijum maksimalne težine ruksaka, pri čemu trenutni podskup T čuvamo ukoliko je profit ostvaren sa T veći od trenutno maksimalnog profita; konfiguracija se predstavlja logičkim nizom: in[k] = true / false → k-ti predmet jeste/nije u ruksaku; dopustivi elementi za k-tu komponentu su: true (ako se k-ti predmet može smestiti u ranac – zbir težina prethodno ubačenih predmeta sa težinom k-tog je manja ili jednaka od maksimalno dozvoljene), ili false (uvek).

**knapsack(int W, int[] wgt, int[] prf, int n)**: provera da li je težina ruksaka W ili broj predmeta n nula (trivijalni slučajevi); provera da li predmet n-1 (predmeti se dodaju od kraja) staje u ranac (wgt[n-1] > W), ako ne rekurzija ide dalje sa n-1, ako da traži se maksimum dve situacije – kada se predmet doda i kada se ne doda.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB03_knapsack/KnapsackG4G.java>

Rešenje problema ruksaka algoritmom grana i granica.

04

**ALGORITAM GRANA I GRANICA (*BRANCH AND BOUND*)**: radi se o modifikaciji *backtracking* postupka za probleme kombinatorne optimizacije; osnovna ideja: 1) kreće se od trenutne konfiguracije sistema Ck-1, gde je Dk skup dopustivih elemenata za k-tu komponentu; 2) za svako d iz Dk ocenjuje se gornje ograničenje (*bound*) f-je f u ciljnim konfiguracijama koje se dobijaju iz Ck-1 proširenim sa d; 3) proširenje Ck-1 sa d se isplati samo ako je bound(d) ≤ trenutnog maksimalnog f.

***BRANCH AND BOUND* REŠENJE RUKSAKA**: postoji neko prethodno ustanovljeno trenutno maksimalno rešenje M; za k-ti predmet i trenutnu parcijalnu konfiguraciju potrebno je utvrditi: **1)** koliko je gornje ograničenje maksimalnog profita PA ubacivanjem predmeta, pod uslovom da se može dodati; **2)** koliko je gornje ograničenje PS neubacivanjem predmeta; proverava se PA > M i ako je tačno, predmet se ubaci i nastavlja se *backtrack*, a ako nije ne ubacuje se i *backtrack* ide dalje; ključno: gornje ograničenje se može ustanoviti gramzivim algoritmom.

**VEZA GRAMZIVOG SA *BRANCH AND BOUND* ALGORITMOM**: problem ruksaka se u slučaju deljivih predmeta lako rešava gramzivim algoritmom; svaka konfiguracija koja se dobija kod 1/0 rešenja ruksaka može se dobiti i rešavanjem gramzivim algoritmom, zato je profit ostvaren selekcijom nedeljivih predmeta uvek manji ili jednak profitu selekcijom deljivih.

**static class Item { .. }**: polja profit, weight; komp. po gustini profita – profit / weight.

**public void find()**: sortira se niz items i poziva privatna metoda find().

**private void find(k,** **currWgt,** **currPrf)**: ako je k-ti (trenutni) Item poslednji i ako je trenutni profit veći od maksimalnog ažurira se maksimalna konfiguracija, inače se proverava da li trenutni Item staje u ruksak (currWgt + item[k].wgt <= knapWgt) i ako da računaju se weight i profit za ruksak sa ubačenim **Item**-om a zatim i **boundAdd** za takvu situaciju – ako je **boundAdd** veći od trenutnog maksimalnog profita čuva se konfiguracija, inače se računa **boundSkip** za situaciju kada **Item** nije ubačen u ruksak i ponovo se poredi sa trenutno maksimalnim profitom.

private double bound(k, weight): prolazak kroz listu **Item**-a od **k**-tog do kraja i sumiranje **weight** i **profit** onih koji staju u ranac; kada se dođe do prvog koji ne staje računa se **fraction** kao količnik preostale težine i težine **Item**-a, a zatim se **fraction** **\*** **item[k].prf** dodaje na sumu profita koja se vraća kao izlaz.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB04_knapsack_bb>

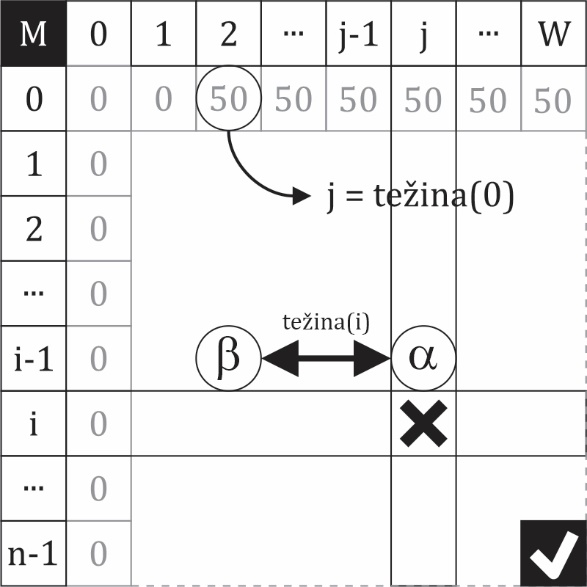
Rešenje problema ruksaka dinamičkim programiranjem.

07

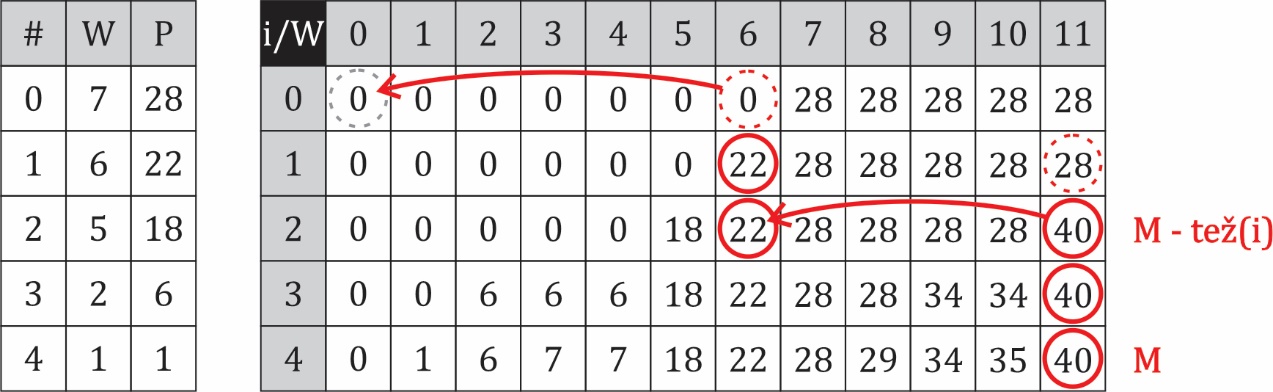
**DINAMIČKO PROGRAMIRANJE**: algoritamski postupak primenjiv na široku klasu problema optimizacije, podrazumeva popunjavanje tabele u procesu rešavanja optimizacionog problema; na ovaj način se rešavaju problemi koji zadovoljavaju *Bellman*-ov princip optimalnosti: problem P ima optimalnu strukturu ako se optimalno rešenje P može dobiti iz optimalnih rešenja manjih instanci problema P (kod ruksaka je ta dimenzionalnost opisana brojem predmeta i kapacitetom ranca); dinamičko programiranje je popunjavanje tabele optimalnih rešenja **svih** instanci problema P(a1, a2, ... , an), nakon čega optimalno rešenje dobijamo iz optimalnih rešenja manjih instanci P zbog čega mora postojati relacija između optimalnog rešenja celog problema i njegovih manjih instanci, a takođe i trivijalne instance od P (kod ruksaka su to situacije sa samo jednim predmetom za proizvoljni kapacitet, i proizvoljan broj predmeta za ruksak kapaciteta 0).

**POSTUPAK**: **1)** kreira se tabela maksimalnog profita sa [0 .. n-1] vrsta (predstavljaju predmete) i [0 .. W] kolona (svi kapaciteti od 0 do W); **2)** posmatrajmo proizvoljni element matrice M[i][j] – ovaj element nam govori koliki profit se može ostvariti ubacivanjem od 0-tog do i-tog predmeta (uključujući i-ti) u ruksak kapaciteta j; **3)** rešenje problema je poslednji element M[n-1][W] koji se dobija ubacivanjem svih predmeta u ruksak kapaciteta W – ideja je da se taj element dobije na osnovu prethodno popunjenih; **4)** rekurzivni odnos se uspostavlja na sledeći način:

Prva opcija je da i-ti predmet ne može da stane u ruksak kapaciteta j tada se optimalno punjenje dobija posmatranjem svih predmeta osim i-tog za ruksak istog kapaciteta; druga opcija je da i-ti predmet može da stane u ruksak i tada se uzima maksimum od profita u slučaju kada se predmet namerno ne uzme i onog u slučaju kada se uzme; nakon postavljanja rekurzivnog odnosa potrebno je odrediti trivijalne slučajeve: **1)** M[i][0] = 0, ∀i ∈ [0, n - 1], **2)** uzmimo npr. da za 0-ti predmet važi težina(0) = 2, profit = 50 – tada se prva vrsta popunjava na način dat u tabeli**:**

****

**REKONSTRUKCIJA STRUKTURE REŠENJA**: rešenje se rekonstruiše posmatranjem tabele maksimalnog profita: ako je MP[i][j] = MP[i -1][j] tada i-ti predmet nije ubačen u ruksak kapaciteta j, a ako nije od trenutne vrednosti profita se oduzima težina trenutnog predmeta i prelazi se na prethodni:



**static class Item { .. }**: polja profit, weight.

**ctor**: tabela mp je dimenzija [items.length][knWeight + 1].

**public int find()**: popunjavanje tabele trivijalnim slučajevima; prolazak kroz tabelu dvema for petljama; računanje profitAdd – ako predmet staje u ruksak kapaciteta trenutne kolone, tada se profitAdd računa kao zbir težine predmeta i polja u prethodnom redu i koloni manjoj za njegovu težinu od trenutne; računanje profitSkip – uzima se vrednost polja iz prethodnog reda ali iste kolone; u tabelu se upisuje veća od te dve vrednosti (lokalno optimalni izbori vode do globalno optimalnog).

**public LL getItems()**: kreiranje brojača i=items.length, j=knapsackWeight i ulazak u while(i > 0); ako važi mp[i][j] != mp[i-1][j] tada se items[i] dodaje u listu a j se smanjuje za težinu dodatog predmeta; na kraju ako je j > 0 i ako je items[0].weight <= j tada se dodaje i items[0].

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB07_knapsack_dp>

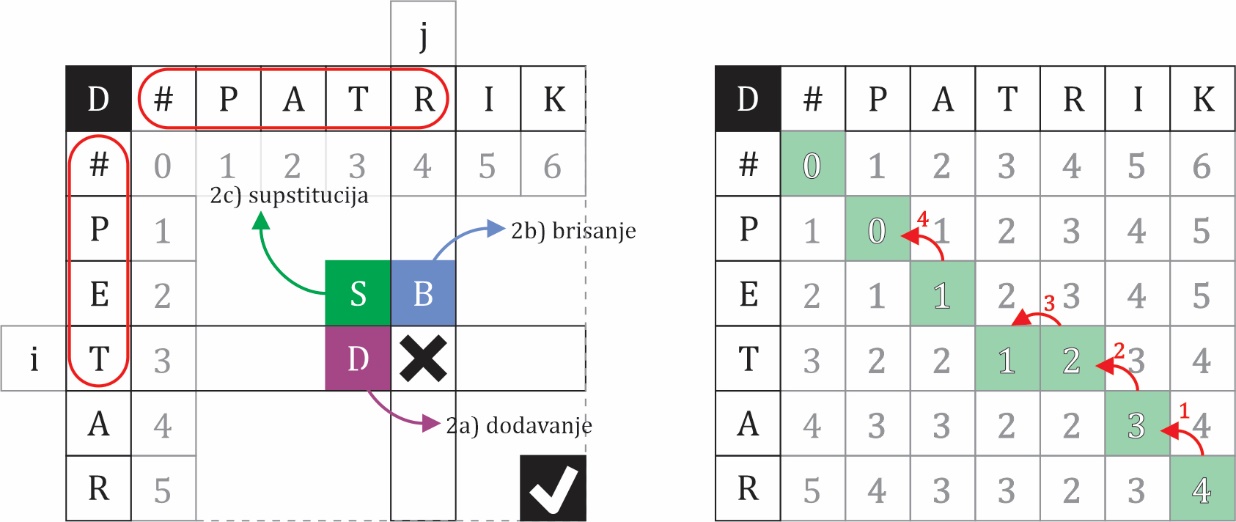
Računanje string *edit* distance dinamičkim programiranjem.

08

**STRING DISTANCE**: reč je o funkciji D(s1, s2), D: String × String → ℝ za koju važi: **1)** D(s1, s2) = 0 → s1 i s2 su identični; **2)** D(s1, s2) = D(s2, s1); **3)** D(s1, s2) < D(s1, s2) → s1 je sličniji stringu s2 nego s3. String distance možemo klasifikovati u: **1)** edit distance – računamo stepen transformacije jednog stringa u drugi, pri čemu veći stepen transformacije znači i veću distancu (odnosno manju sličnost); **2)** set distance – računamo stepen preklapanja stringova, pri čemu veći stepen preklapanja znači i manju distancu (veću sličnost).

**N-GRAM DISTANCE**: klasa jednostavnih set string distanci; postupak: **1)** od stringa se formira skup n-grama (primer: „algoritam“ → „alg“, „lgo“, … , „ita“, „tam“), **2)** sličnost se meri kao stepen preklapanja n-gram skupova stringova za neko fiksirano n; računanje *Jaccard*-ove n-gram distance:

**EDIT STRING DISTANCA**: najjednostavnija edit string distanca meri broj elementarnih transformacija potrebnih da se jedan string transformiše u drugi, pri čemu su elementarne operacije dodavanje, brisanje i supstitucija karaktera; računanje edit distance je optimizacioni problem – traži se minimalni broj elemenata transfromacija (primer: „petar“ → „patrik“, 4 transformacije).



Tarabom označene prva kolona i prva vrsta služe da bi se mogla popuniti tabela za trivijalne slučajeve; Element D[i][j] govori kolika je edit distanca od stringa „pet“ do stringa „patr“, odnosno od prvih i karaktera prvog i prvih j karaktera drugog stringa (koji su indeksirani ne od 0 nego od 1); uopšteno, D[i][j] = edit(a1a2...ai, b1b2...bj); postavlja se rekurentna relacija:

**1)** prvi slučaj – ai = bj (recimo „petar“ i „patr“), tada je D[i][j] = D[i-1][j-1], čime se situacija redukuje na problem manje dimenzionalnosti;

**2)** drugi slučaj – postoje tri opcije:

**a)** na kraj prvog stringa se dodaje poslednji karakter drugog, pa a1a2...ai, b1b2...bj postaje a1a2...aibj, b1b2...bj (*insert* transformacija - **D**), nakon čega je D[i][j] = 1 + D[i][j-1],

**b)** sa kraja prvog stringa se briše i-ti karakter, pa a1a2...ai, b1b2...bj postaje a1a2...ai-1, b1b2...bj (*delete* transformacija - **B**) , nakon čega je D[i][j] = 1 + D[i-1][j],

**c)** poslednji karakter prvog stringa se zameni poslednjim karakterom drugog, pa a1a2...ai, b1b2...bj postaje a1a2...bj, b1b2...bj (*substitution* transformacija - **S**), nakon čega je D[i][j] = 1 + D[i-1][j-1],

Rešenje se rekonstruiše „šetnjom“ od poslednjeg do prvog elementa matrice (od „patrik“ do „petar“: **1)** supstitucija „k“ u „r“, **2)** supstitucija „i“ u „a“, **3)** brisanje „r“, **4)** supstitucija „a“ u „e“; tamo gde je minimum jednak trenutnoj vrednosti nije bilo promena.

**computeDistance()**: popunjavanje tabele d[][] za trivijalne slučajeve (po jedna for petlja za horizontalu i vertikalu); kretanje kroz matricu pomoću dve for petlje, uzimanje tekućeg karaktera za oba stringa (s1.charAt(i), s2.charAt(j)); ako su karakteri jednaki, polje uzima vrednost d[i-1][j-1], ako nisu ispituje se koja je od transformacija izvedena i u zavisnosti od toga polje uzima 1 + minimum od tri vrednosti: d[i-1][j] (brisanje), d[i-1][j-1] (supstitucija) ili d[i][j-1] (dodavanje).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB08_string_distance>

Opšte binarno stablo, broj čvorova i dubina binarnog stabla, iteriranje kroz binarno stablo.

09

**STABLO**: struktura podataka kojom se može predstaviti hijerarhijsko uređenje skupa, odnosno stablo je hijerarhijski uređena kolekcija elemenata; osnova hijerarhije je relacija „otac – sin“, pri čemu svaki sin može imati najviše jednog oca; stablo se sastoji od čvorova i grana koje spajaju čvorove: **1)** čvorovi A i B su povezani granom ukoliko je A otac od B; **2)** stablo ima jedinstven čvor bez oca – koren; **3)** svi čvorovi osim korena imaju tačno jednog oca; **4)** svi čvorovi imaju proizvoljan broj sinova.

**BINARNO STABLO**: binarno stablo je stablo kod kog svaki čvor ima najviše dvoje dece, što je oblik kojim se može predstaviti svako stablo (primer); ono se može predstaviti nizom (kao u slučaju prioritetne liste), gde su sinovi čvora na k-toj poziciji s1 = 2k + 1 i s2 = 2k + 2, dok je roditelj čvora na k-toj poziciji r = (k – 1) / 2 (celobrojno) – problem kod ovakve reprezentacije nastaje kada je stablo retko, odnosno kada ima mnogo manje od M = 2k+1 – 1 čvorova, što je maksimalan broj čvorova za stablo dubine k; dinamička reprezentacija binarnog stabla podrazumeva čuvanje pokazivač na oba sina unutar čvora (opciono i na roditelja), dok stablo čuva referencu na koren.

**Konstruisanje i modifikovanje**: opšte binarno stablo (koje nema neke posebne osobine) se konstruiše tako što se kreiraju čvorovi stabla i eksplicitno postave veze sa setLeft() ili setRight() metodama, nakon čega jedan element proglasimo za korenski; stablo se modifikuje tako što se menjaju vrednosti referenci čvorova i/ili vrednost reference na koren.

Karakteristike: dve osnovne karakteristike su veličina (broj čvorova u stablu) i dubina; za svaki čvor binarnog stabla postoji tačno jedan put od čvora ka korenu, a dubina je dužina najvećeg puta od proizvoljnog čvora stabla do korenskog elementa (ako stablo ima samo koren dubina je 0, a ukoliko je prazno stablo je dubine -1).

**class BTNode<T extends Comparable<T>> implements Comparable<BTNode<T>>**: ctor-i (info bez čvorova i sa čvorovima), get & set metode, toString i compareTo.

**class BinaryTree<T extends Comparable<T>>**: od “osnovne opreme” sadrži polje root, get & set za root, isEmpty, size i depth metode.

**size()**: public metoda proverava da li je root nula i vraća 0 ako jeste, inače ulazi u private metodu od root-a; private metoda proverava da li je levi sin null, ako nije vraća rekurzivni poziv size() za left, isto tako i za desnog sina, na kraju se vraća zbir te dve vrednosti + 1.

**depth()**: postupak identičan onom u metodu size() s tim da se na kraju dve vrednosti porede i vraća se veća + 1.

**dfs(T info)**: postupak sa public i private metodom identičan prethodnim; private metoda prima node i info – provera da li je info u čvoru jednak traženom i vraćanje čvora ako jeste, inače provera da li postoji levi sin i rekurzivni poziv dfs() metoda za levi čvor uz vraćanje čvora ako je info pronađen, identičan postupak za desnog sina, ultimativno vraćanje null.

**bfs(T info)**: *breadth first search* pretraga se implementira rekurzivno; provera da li je root == null, ako nije kreira se queue, dodaje root i ulazi u while petlju koja se izvršava dok queue nije prazan; unutar petlje se izbacuje i uzima prvi čvor iz queue, provera da li sadrži info i ako da vraća se čvor, inače se proverava da li ima levog sina i njegovo ubacivanje u queue ako da, isto za desnog; ultimativno vraćanje null.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB09_10_11_binary_tree>

Opšte binarno stablo, pretraživanje binarnog stabla u dubinu.

10

**DEPTH FIRST SEARCH**: svaki čvor binarnog stabla sadrži neku informaciju – pretraga podrazumeva proveru da li postoji čvor koji sadrži informaciju „x“; pretraživanje u dubinu se odvija po sledećem postupku: **1)** da li je „x“ sadržan u korenu; **2)** ako nije, traži u levom podstablu koristeći isti postupak (rekurzivno); **3)** ako nije, traži u desnom podstablu.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB09_10_11_binary_tree>

Opšte binarno stablo, pretraživanje binarnog stabla u širinu.

11

**BREADTH FIRST SEARCH**: realizuje se koristeći red opsluživanja kako bi se postiglo pretraživanje po nivoima; postupak: **0)** doda se koren u red; **1)** iz reda se ukloni prvi element – ako je to tražena informacija postupak je završen; **2)** ako nije, proveri se da li postoji levi sin i doda se u red ako postoji, zatim se isto uradi sa desnim sinom; **3)** prethodna dva koraka se ponavljaju sve dok se red ne isprazni.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB09_10_11_binary_tree>

ADT skup realizovan binarnim stablom pretraživanja, operacije *insert* i *member*.

12

**BINARNO STABLO PRETRAŽIVANJA**: kod BST-a čvorovi su jedinstveni i za svaki čvor stabla A važi da je veći od svih elemenata u levom podstablu, i manji od svih u desnom podstablu. Skupovi se mogu realizovati binarnim stablom pretraživanja i u tom slučaju su elementarne operacije poput insert, search, remove u proseku logaritamske složenosti.

**class BTNode**: čvor identičan onom kod binarnog stabla.

**PRETRAŽIVANJE**: **1)** provera da li se info nalazi u korenu stabla – ako da, kraj; **2)** ako je info manje od koren.info tada se traženi element traži samo u levom podstablu (rekurzivno na isti način), a ako je veće radi se isti postupak za desno podstablo; za balansirano stablo složenost je O(log n) budući da se veličina problema u svakom koraku polovi; pretraživanje BST-a je šetnja kroz stablo od korena ka listovima gde se iz svakog čvora ide ili levo ili desno.

**search (T info)**: koriste se curr (incijalizovan na root) i prev (null) čvorovi, kao i found flag (false); kretanje kroz while petlju dok curr != null i !found; kreira se double cmp sa vrednošću poređenja curr.info i info – ako je == 0 tada se found postavlja na true, inače prev uzima vrednost curr a curr uzima vrednost left (ako je cmp < 0) ili right; na kraju se rezultat pretrage vraća kao objekat unutrašnje klase SearchResult koja sadrži polja node i parent, kao i odgovarajući ctor.

**DODAVANJE**: uvek je reč o kreiranju novog lista BST-a; postupak: **1)** ako je stablo prazno kreira se novi korenski čvor sa info; **2)** ako stablo nije prazno, kreće se u šetnju kroz stablo od korena ka listovima na sledeći način: ako je info manje od informacije u tekućem – idi levo, inače – idi desno; **3)** ako levi, odnosno desni sin ne postoje nakon prethodnog koraka novi čvor koji sadrži info postaje levi, odnosno desni sin tekućeg čvora; **4)** ukoliko trenutni čvor sadrži info dodavanje se obustavlja.

**member (T info)**: provera da li je pretraga stabla za info nalazi takav čvor.

**insert (T info)**: provera da li je root == null i ako jeste kreiranje novog čvora i postavljanje njega za root; kreiranje SearchResult objekta za info – ako node polje != null info već postoji u stablu i vraća se false; kreiraju se newNode (sadrži info) i parent (polje SearchResult-a) čvorovi i porede se – ako je double cmp > 0 novi čvor je veći od oca i smešta se desno od njega, inače levo od njega i vraća se true u oba slučaja.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB12_13_bintree_adt_set>

ADT skup realizovan binarnim stablom pretraživanja, operacija *delete*.

13

**BRISANJE**: da bi se obrisala informacija info iz stabla ono se pretražuje i traži se čvor koji sadrži info; postoje tri slučaja: **1)** info je u listu – trivijalno; **2)** info je u unutrašnjem čvoru sa jednim sinom – prosto prevezivanje kao kod JPL; **3)** info je u unutrašnjem čvoru koji ima oba sina – nešto složenije; dodatno, vodi se računa o tome da li je čvor koji brišemo koren.

**remove (T info)**: SearchResult za info, ako je node == null vraća false; posmatra se node – ako su oba sina == null tada se radi o brisanju lista, ako je (samo) jedan od njih null radi se o brisanju unutrašnjeg sa jednim sinom, inače se radi o brisanju unutrašnjeg; vraća se true u svakom slučaju.

**Brisanje lista**: ako je parent == null, tada se uklanja korenski element; inače se proverava da li je čvor za brisanje levi ili desni sin i odgovarajući pokazivač postavlja se na null.

**removeLeaf (BTNode<T> parent, BTNode<T> toRemove)**: provera da li je parent == null – ako jeste znači da se uklanja root i tada se prosto on postavi na null, inače se proverava da li se uklanja levi ili desni sin parent čvora i odgovarajući čvor se postavi na null.

**Brisanje čvora sa jednim sinom**: pronalazi se child - sin čvora koji se briše; ako je parent == null tada se postavlja child za koren; inače se proverava da li se briše levi ili desni sin i odgovarajući pokazivač se prevezuje na child.

**removeInternalWithOneChild (parent, BTNode<T> toRemove)**: na početku je ista provera kao i u prethodnom slučaju; kreira se čvor child koji sadrži levog ili desnog sina toRemove čvora (u zavisnosti kojeg sina ima), zatim se proverava da li je toRemove levi ili desni sin čvora parent i child se tada prevezuje na isto mesto gde je bio toRemove.

**Brisanje unutrašnjeg čvora**: osnovna ideja je da se ne briše sam čvor C već da se obriše neki čvor koji ima jednog sina ili koji je list, a da se informacija čuvana u obrisanom čvoru smesti u C; postoje dve implementacije:

**1)** pronalazi se M – najmanji čvor u desnom podstablu čvora C (M ne može imati levog sina jer onda ne bi bio najmanji, a ako zamenimo C sa M očuvana je osobina BST-a – s jedne strane, M je veći od svih čvorova u levom podstablu C jer je M > C a svi čvorovi u levom podstablu C su manji od C, a sa druge M je najmanji od svih čvorova u desnom podstablu C);

**2)** pronalazi se najveći čvor u levom podstablu čvora C.

Razmotrićemo prvu implementaciju: najmanji čvor u desnom podstablu čvora C se pronalazi na sledeći način: iz C se ide jednom udesno, a zatim ulevo koliko je god moguće:

min = C.getRight();

while (min.getLeft() != null) min = min.getLeft();

Pored toga, potreban je i otac najmanjeg čvora radi prevezivanja ako min nije list.

**removeInternal (parent, BTNode<T> toRemove)**: kreiraju se čvorovi min (inicijalizovan na toRemove.right()) i minParent (toRemove), korišćenjem while petlje se nađe minimum levog podstabla na način opisan iznad; preuzima se min.info i taj čvor se ukloni – u zavisnosti od toga da li je napravljen bar jedan korak ulevo ili nije izvršava se minParent.setLeft(null) ili .setRight(null); konačno, toRemove.info se postavlja na (sačuvanu) vrednost min.info.

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB12_13_bintree_adt_set>

Konstrukcija Hafmanovog stabla kodiranja – opis postupka.

14

**KOMPRESOVANJE TEKSTA**: pojmovi: **1)** kod – bitstring, binarna sekvenca koja odgovara simbolu; **2)** kodni sistem – 1-1 preslikavanje skupa simbola u skup kodova, pri čemu su kodovi iste dužine (primer: *ASCII* kodni sistem, gde svakom karakteru odgovara 8-bitni bitstring) čega je prednost jednostavno (de)kodiranje. **Osnovna ideja kompresovanja** teksta sastoji se od: **1)** kreiranja novog kodnog sistema uzimajući u obzir samo karaktere koji se pojavljuju u tekstu, **2)** pravljenja novog kodnog sistema promenljive dužine, pri čemu simboli sa većom frekvencijom pojavljivanja imaju kod manje dužine, što se obezbeđuje **3)** prefiksnim kodovima kako bi dekompresija bila moguća (budući da nijedan kod nije prefiks drugog koda):

neprefiksni sist.: {а→0, b→1, c→01, d→11}, 010111 = “ababbb” / “ccbb” / “ccd”…

prefiksni sist.: {a→0, b→10, c→110, d→111}, 010111 = “abd” jednoznačno

Prefiksni kod se može predstaviti binarnim stablom gde su listovi simboli, a kod za simbol je put od korena ka simbolu. Suština kompresije je minimizovanje f-je , gde je n broj različitih karaktera u tekstu, fi frekvencija i-tog karaktera, a di dužina koda za i-ti karakter (dubina odgovarajućeg lista u binarnom stablu kodiranja); zato listovi blizu korena treba da imaju veliku frekvenciju.

**HAFMANOVO STABLO KODIRANJA**: kod ovakvog stabla svaki čvor kao informaciju nosi frekvenciju (listovi još i karaktere), a frekvencija unutrašnjeg čvora jednaka je zbiru frekvencija njegove dece. Postupak: **1)** napravimo čvorove koji su listovi stabla i dodamo ih u skup S; **2)** neka su A i B dva čvora sa najmanjim frekvencijama u S; **3)** napravimo novi čvor C tako da C.right = A, C.left = B, C.freq = A.freq + B.freq; **4)** A i B obrišemo iz S, dodamo C; ponavljamo **2)**, **3)** i **4)** dok ne ostane jedan čvor koji postaje koren stabla (primer). Skup S može se realizovati:

1) **sortiranom listom**: **1)** listove na početku treba sortirati po frekvencijama – O(n logn), **2)** selekcija i brisanje dva čvora sa minimalnim frekvencijama – O(1), **3)** dodavanje novog čvora u S – O(n); ukupno – O(n2), jer korake **2)**, **3)** radimo (n-1) puta;

2) **prioritetnom listom**: **1)** listove na početku treba dodati u listu – O(n logn), **2)** selekcija i brisanje dva čvora sa minimalnim frek. – O(log n), **3)** dodavanje novog čvora u S – O(log n); ukupno – O(n logn).

**class CharFrequency implements Comparable<CharFrequency>**: klasa objekti koje se nalaze u čvorovima stabla; sadrži polja char c i int freq, kao i ctor; poređenje je po frekvenciji.

**public HuffmanTree(String inputText)**: pozivaju se metodi computeFrequencies(inputText) i construct().

**computeFrequencies(String inputText)**: inicijalizuje se polje klase frequencyList (lista koja sadrži CharFrequency objekte), kreira se HashMap frequencyIndex koja sadrži parove (char, CharFrequency); for petljom se prolazi kroz sve karaktere ulaznog stringa i za svaki karakter se preuzima odgovarajući CharFrequency objekat iz HashMap-e – ako takav objekat ne postoji kreira se nov za dati char i freq = 1, zatim se CharFrequency dodaje u frequencyList, kao i par (char, CharFrequency) u HashMap; ukoliko takav objekat postoji freq polje unutar CharFrequency se inkrementira.

**construct()**: uzima se broj elemenata frequencyList (odnosno broj različitih karaktera u ulaznom tekstu) za veličinu PriorityQueue<CharFrequency>, frequencyList se pretvara u iterator; prolazak kroz frequencyList iterator while petljom i kreiranje čvorova sa karakterima iz liste; prolazak kroz elemente PriorityQueue korišćenjem while: uzimaju se prva dva čvora iz liste, kreira se njihov roditelj sa CharFrequency(‘#’, freqL+freqR), povezuju se sa roditeljem i roditelj se vraća u PriorityQueue – while petlja se završava kada je pq.size() == 1; stablo hTree se tada kreira i za root uzima jedini element PriorityQueue.

**printCodes()**: uzima se root stabla i proverava se da li su mu oba sina == null – ako da, štampa se njegov kod „1“, inače se ulazi u istoimenu private metodu (parametri BTNode<> curr i String code); provera da li su oba sina curr == null, ako jesu ispisuje se code za karakter u curr, ako nisu rekurzivno se poziva private metoda i to za levog sina sa parametrima (curr, code + „0“) a za desnog sa (curr, code + „0“).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB14_huffman_tree>

Podeli i osvoji – opšti opis postupka i matematička tvrđenja.

15

***DIVIDE AND CONQUER***: „podeli i osvoji“ je rekurzivan algoritamski postupak kod kog se **1)** polazni problem podeli na više manjih nezavisnih podproblema iste vrste – **2)** ako je taj podproblem trivijalan („mali“) rešava se direktno, inače se rešava rekurzivno (razbijanjem na još manje podprobleme), da bi se na kraju **3)** rešenje polaznog dobilo objedinjavanjem rešenja podproblema; najčešće je reč o podeli na dva podproblema. **Primeri**: merge sort (podeli: podela na dve podliste, osvoji: spajanje sortiranih podlisti), quick sort (podeli: podela na dve podliste spram pivota, osvoji: kalemljenje sortiranih podlisti) – u oba slučaja trivijalni podproblem je elementarna ili mala lista koja se sortira elementarnim metodama.

Pseudo-kod opšteg slučaja „podeli i osvoji“:

if (p is an elementary problem)

**solve p directly** and return solution

else {

divide p into independent

problems p1 and p2

Solution **s1 = solve(p1)**;

Solution **s2 = solve(p2)**;

merge s1 and s2 into s where s is solution of p;

return s }

**„PODELI I OSVOJI“ TEOREMA**: neka se polazni problem veličine n deli na dva podproblema veličine n/2 i neka se trivijalni podproblem rešava u konstantnom vremenu; vremenska složenost celog postupka može se izraziti sledećom rekurentnom relacijom: T(n)=2T(n/2) + f(n), gde je f(n) vreme potrebno za podelu problema i objedinjavanje rešenja. Teorema (čiji je dokaz direktna posledica master teoreme):

1) ako je f(n) u klasi O(1) tada je T(n) u klasi O(n);

2) ako je f(n) u klasi O(n) tada je T(n) u klasi O(n logn);

3) ako je f(n) u klasi O(n2) tada je T(n) u klasi O(n2).

**MASTERTEOREMA**: neka je data rekurentna relacija T(n)=aT(n/b) + f(n), gde je f(n) u O(nd) (T(1) je u O(1)), i gde su a, b, d konstante takve da a > 0, b > 1, d ≥ 0; tada je:

1) T(n) u O(nd) ako je d > logba;

2) T(n) u O(nd logn) ako je d = logba;

3) T(n) u O(n^logba) ako je d < logba.

**Primer**: za T(n)=2T(n/2) + f(n), imamo a=b=2 → logba=1:

1) ako je f(n) u O(1) tada d = 0 < logba → T(n) u O(n);

2) ako je f(n) u O(n) tada d = 1 = logba → T(n) u O(n logn);

3) ako je f(n) u O(n2) tada d = 2 > logba → T(n) u O(n2);

**BIN. PRETRAŽIVANJE I STEPENOVANJE UZASTOPNIM KVADRIRANJEM**: oba problema su takođe algoritmi iz klase „podeli i osvoji“ i kod oba se početni problem veličine n redukuje na jedan podproblem veličine n/2, gde se rešenje polaznog (osvajanje) trivijalno dobija iz rešenja redukovanog problema; za oba važi da je T(n)=T(n/2) + f(n), gde je f(n) u O(1), pa primenom Master teoreme dobijamo: a = 1, b = 2 → logba = 0; d = 0 = logba → T(n) je u O(nd logn); drugim rečima T(n) je u O(logn) jer je n0 = 1.

**MNOŽENJE KVADRATNIH MATRICA**: neka su A i B dve kvadratne matrice dimenzije n, tada je množenje po definiciji u O(n3); „podeli i osvoji“ pristup – A i B se podele u blokove dimenzija n/2:

Štrasen je pokazao da se blokovi matrice C mogu dobiti iz blokova matrica A i B uz 7 množenja i 16 sabiranja: T(n)=7T(n/2) + f(n), gde je f(n) = 16(n/2)2 → f(n) u O(n2); primenom Master teoreme dobijamo a = 7, b = 2, logba = 2.807, d = 2 → d < logba → T(n) u O(n^ logba) = O(n2.807); blokove trivijalno dobijamo iz 8 množenja i 4 sabiranja, no tada je T(n) u O(n3).

Podeli i osvoji – rešenje problema najkraće distance u skupu 2D tačaka.

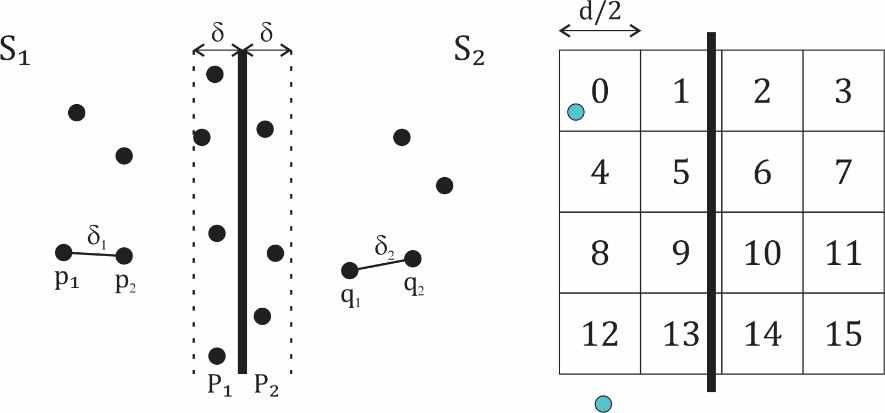
16

**NAJKRAĆA DISTANCA U SKUPU 2D TAČAKA**: jedan od osnovnih problema računarske geometrije je određivanje najkraćeg rastojanja između dve tačke niza tačaka u ravni S; naivan pristup je u O(n2) – prosto računanje distance između svake dve tačke tražeći minimum (primer).

**„PODELI I OSVOJI“ PRISTUP**: niz S „podeli se“ na dva dela – S1 = S[0 .. n/2] i S2 = S[n/2+1 .. n-1], n = S.length; uzimaju se m1/m2 = minimalna distanca u S1/S2 , i m3 = minimalna distanca između tačaka A i B, gde A ∈ S1, B ∈ S2; ako bi se m3 tražio tako što se računa rastojanje svake dve tačke A i B, gde A ∈ S1, B ∈ S2 tada bi važilo:

T(n)=2T(n/2) + f(n), gde je f(n) iz O(n2) → T(n) je u O(n2)

Pametniji “podeli i osvoji” pristup podrazumeva da se S sortira po x koordinati, nakon čega se izvrši podela na S1 i S2:



Kod ovog pristupa  = min(m1, m2) – poenta je napraviti niz C u kome su sadržane sve tačke koje su od tačke S[n/2] po x koordinati udaljene manje od ; niz C se sortira po y koordinati i tada za svaku tačku iz C najviše 15 sledećih tačaka može biti na rastojanju manjem od : budući da je minimalno rastojanje u obe polovine niza  važi

**Kompleksnost pristupa**: T(n)=2T(n/2) + f(n), gde je f(n) komponovan od sledećih operacija: **1)** formiranje niza C – O(n), **2)** sortiranje niza C po y koordinati – O(n logn), **3)** računanje distanci za tačke iz niza C – O(1) (distance se računaju za najviše 15 sledećih tačaka); stoga je f(n) u O(n logn), pa je T(n) tada u O(n log2n), početno sortiranje po x koordinati je u O(n logn) – stoga je vremenska složenost algoritma O(n log2n).

**class** **Point2D**: klasa koja opisuje 2D tačku; metoda coincident(Point2D p) proverava da li su pozicije tačaka poklapaju, metoda dist(Point2D p) računa distancu između dve tačke.

**class** **MinDistance2D**: klasa sadrži niz Point2D[], ugnježdene komparatore po x i y koordinatama i ctor koji prima niz tačaka.

**findDQ()**: public metoda koji sortira tačke po x koordinati, proverava da li postoje poklapajuće i vraća minimalnu distancu 0.0 ako da, inače poziva istoimenu private metodu.

**findDQ(int** **from,** **int** **to)**: na početku provera za dva posebna slučaja – da li niz sadrži jednu tačku (minDist == Double.MAX\_VALUE) i da li su **from** i **to** zamenili mesta; računa se idx median, rekurzivno se poziva findDQ(from, median) i findDQ(median+1, to) i računa se minDist – manja od dve vrednosti (koja će postati delta parametar); konačno, poziva se minCrossing(from, to, median, delta) i vraća se Math.min(minDist, minCrossing).

**minCrossing(int** **from,** **int** **to,** **int** **median,** **double** **delta)**: traži se x koordinata vertikale koja deli niz tačaka, kreira se Point2D[] box kuda se smeštaju tačke koje su po x na rastojanju od vertikale manjem od delta; niz se sortira komparatorom po y i to od idx 0 do boxCount, nakon čega se ulazu u for petlju od 0 do boxCount - 1; uzima tačka box[i] i pokreće nova for petlja od i + 1 do boxCount (ipak, ova petlja će se izvršiti najviše 15 puta – razlog: <https://tinyurl.com/y6nabnha>) tokom koje se proverava da li je udaljenost po y trenutne od tačke box[i] veća od delta – ako jeste, petlja se prekida, inače se proverava udaljenost tačke od box[i] i ako je udaljenost manja od trenutno minimalne minDist uzima dobijenu vrednost; konačno, vraća se minDist.

Dodatno: <https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/>

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB16_string_distance>

Pretraživanje stringova grubom silom i RK algoritmom.

17

**BRUTE FORCE**: ideja je da se za svaki podstring teksta proveri da li je jednak *pattern*-u: **1)** poravna se *pattern* sa početkom stringa tekst, **2)** proveri se da li se poravnati znaci poklapaju, **3)** ako da – pronađena je prva pojava *pattern*-a u tekstu, **4)** kod prvog nepoklapanja poravnatih znakova *pattern* se pomera za jedno mesto udesno i postupak se vraća na korak 2).

**RABIN-KARPOV ALGORITAM**: ideja je da se za svaki string izračuna *hash* kod koji se tada koristi kao *fingerprint* (*signature*) za preliminarno poređenje stringova – uzimamo da je HP *hash* *pattern*-a a HT *hash* teksta (za proizvoljno poravnanje računa se *hash* kod poravnanjem obuhvaćenog podstringa tekst); ako se HP i HT razlikuju tada se sigurno za da nema poklapanja svih poravnatih karaktera, u suprotnom se poredi znak po znak kao kod *brute force*  pristupa; *hash* kod za klasu String računa se na sledeći način:

hash = s[0]\*31L + s[1]\*31L-1 + s[2]\*31L-2 + .. + s[L-1]\*31 + s[L], L = length() – 1

Trik kod RK algoritma je u tome što ako bi se za svaki podstring teksta računao *hash* algoritam ne bi bio efikasniji od *brute force* pristupa (složenost O(P\*T), gde je P dužina pattern-a a T teksta) – ovaj problem se rešava ažuriranjem *hash* koda podstringa teksta u O(1) nakon što se napravi novo poravnanje tako što se briše prvo slovo i dodaje novog na kraj podstringa; vrednost *hash* f-je se ograničava u opsegu [0 .. LP] gde je LP neki veliki prost broj kako bi se moglo izvesti ažuriranje (budući da 31L može biti veće od Integer.MAX\_VALUE za dugačak *pattern*).

**Ažuriranje *hash* vrednosti**: neka je String s sačinjen od karaktera C0, C1, .. , CL – *hash* je tada:

hash(s) = (C0\*31L + C131L-1 + .. + CL-1\*31 + CL) % LP

*Hash* kod podstringa nakon pomeranja možemo dobiti iz hash(s) na sledeći način:

X = (hash(s) – (C0\*(31L % LP)) % LP) % LP (vrednost *hash*-a kada se iz s odstrani C0)

hash(s) može biti manje od Y = (C0\*(31L % LP)) % LP: X = (**LP** + hash(s) – Y) % LP

Konačno, X se množi sa 31 i dodaje se B (novi karakter na kraju podstringa nakon pomeranja):

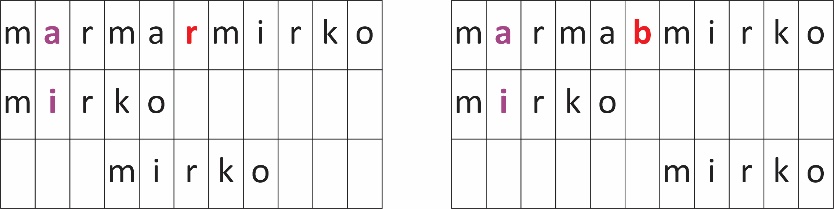
hash(s’) = ((31\*X) % LP + B) % LP (31L % LP je konst. koja se računa samo na početku)

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB17_bf_rk_search>

Pretraživanje stringova Quick search algoritmom.

18

**QUICK SEARCH**: neka je dato poravnanje *pattern*-a P sa tekstom T i neka je q probni znak, odnosno prvi znak T nakon poravnanja; porede se karakteri P sa karakterima iz T u tekućem poravnanju *brute force* pristupom – kada dođe do neslaganja karaktera u prethodnom koraku tada se ili **1)** pomera P tako da se q iz T poklopi sa poslednjom pojavom q u P, **2)** a ako se q ne pojavljuje u P tada se P pomera iza q (u ovom slučaju pomeraj je jednak dužini P+1):



U prvom slučaju P se pomera u zavisnosti od pozicije q u *pattern*-u: za jedan ukoliko je q poslednji karakter P, za dva ukoliko je q pretposlednji, itd.; formira se mapa pomeraja u kojoj je ključ neki karakter iz P, a vrednost dužina pomeraja P za taj karakter – ova mapa se jednostavno formira obilaskom stringa od kraja ka početku, pri čemu treba voditi računa da u P neki znak može da se pojavi više puta (traži se poslednja pojava).

<https://github.com/NikolaVetnic/SPA2/tree/master/src/p_vezba/qB18_quick_search>

1. Drugim rečima, *Comb* prolaz se za k = 1 ponavlja dok postoje elementi u inverziji što je zapravo *Bubble sort*. [↑](#footnote-ref-1)
2. Kolizije i “*birthday paradox*”: 367 ljudi u sobi za 100% verovatnoće da su dve rođene istog dana, 70 za 99,99%, 35 za 80%, 23 za 50%. [↑](#footnote-ref-2)
3. Praktično slaže sva polja prve vrste u niz od poslednjeg ka prvom. [↑](#footnote-ref-3)
4. Ovo je važno ukoliko dođe do povratka na ovo polje jer će se tako isprobati drugačije polje nego prvi put [↑](#footnote-ref-4)