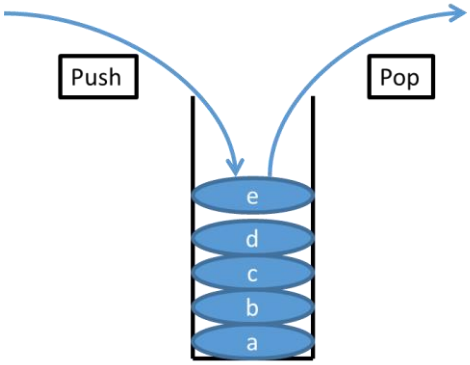
Stog, red i prioritetni red

Ovdje ćemo opisati i implementirati neke jednostavnije apstraktne tipove podataka (ATP) koji su česti u računarstvu. Apstraktni tip podatka je kombinacija samog predstavljanja podataka i potrebnih operacija nad njima. Implementacija takvog apstraktnog tipa određuje konkretnu strukturu podataka, izvedbu operacija, kao i složenost takvih operacija.

Stog

Stog je ATP koji uz osnovne operacije poput stvaranja i uništavanja, imaju dvije glavne operacije: *push* i *pop*. Stog je lista koja funkcionira po Last-In-First-Out principu, odnosno zadnji element koji dodamo u listu push operacijom će se prvi izvući iz liste pop operacijom. Po tom principu funkcionira i programski stog, a kasnije ćemo vidjeti još neke primjene stoga.



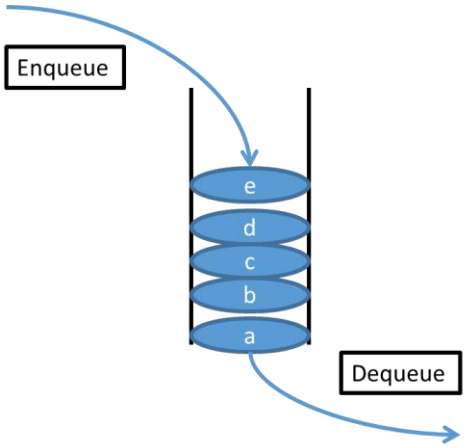
Stog možemo jednostavno implementirati sa nizom ili sa jednostruko vezanom listom jer obje glavne operacije izvodimo na „vrhu“ stoga. U slučaju niza, najefikasnije je vrhom stoga smatrati krajem niza. U slučaju vezane liste, najefikasnije je vrhom smatrati glavu liste. U oba slučaja će obje operacije biti konstantne složenosti.

Red

Red je First-In-First-Out ATP često korišten za realizaciju međuspremnika (*buffera)*, na primjer za obradu elemenata po redu stizanja i slično. Uz osnovne operacije, glavne operacije su *enqueue* i *dequeue* za dodavanje i uklanjanje elemenata sa reda.

```c
typedef struct {
	int first, last, n, size;
	int *buffer;
} Queue;

Queue *create(int size) {
	Queue *q = (Queue*)malloc(sizeof(Queue));
	q->first = q->last = q->n = 0;
	q->size = size;
	q->buffer = (int*)malloc(sizeof(int)*size);
	return q;
}

void enqueue(Queue *q, int b) {
	if (q->n >= q->size)
		return;
	q->buffer[q->last] = b;
	q->last = (q->last + 1) % q->size;
	q->n++;
}

int dequeue(Queue *q) {
	if (q->n <= 0)
		return -1;
	int e = q->buffer[q->first];
	q->first = (q->first + 1) % q->size;
	q->n--;
	return e;
}
```

Implementacija sa nizom se realizira kao cirkularni buffer, gdje je maksimalni kapacitet reda fiksan, ali nanovo iskorištavamo indekse niza koji su oslobođeni uklanjanjem elemenata sa reda. To se postiže tako da pratimo početak reda, kraj reda i broj elemenata u redu. Kada sa indeksom početka ili kraja reda dođemo do fizičkog kraja niza, tada se taj indeks vraća na početak niza (indeks 0). Zbog toga više nema garancije da će indeks početka reda biti manji od indeksa kraja reda i potreban nam je dodatni brojač za broj elemenata u redu da bi mogli odrediti da li je red prazan ili pun.



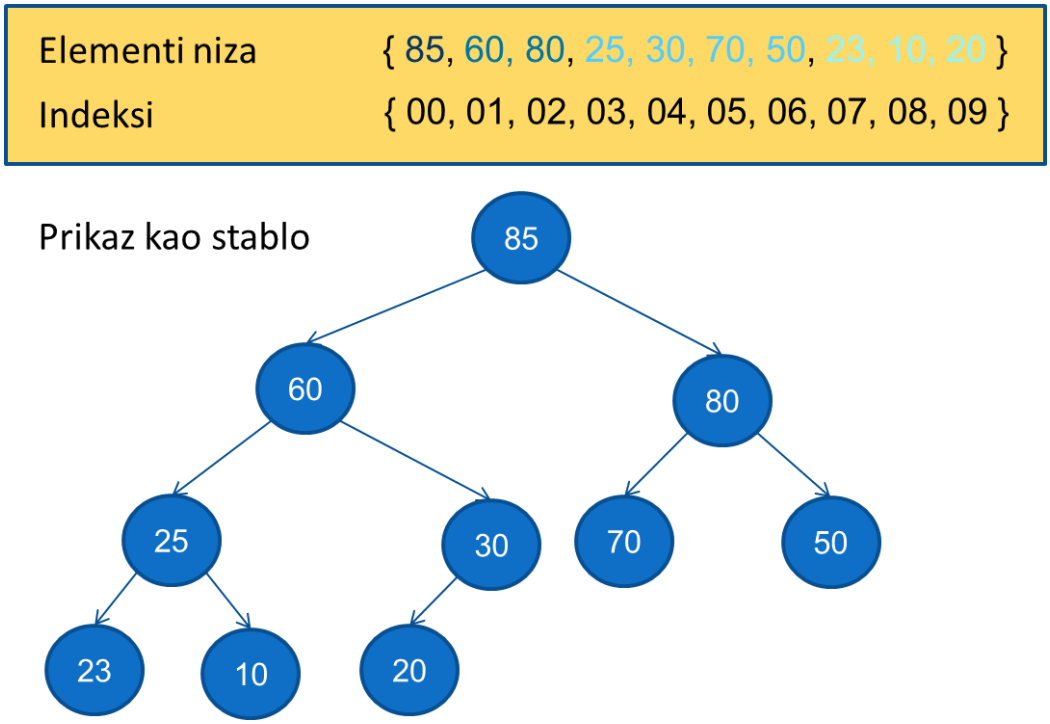
Implementacija sa jednostruko vezanom listom dodaje elemente na kraj liste, a uklanja ih sa početka liste. Na taj način možemo realizirati obje operacije u O(1) vremenu pod uvjetom da imamo dodatni pokazivač na zadnji element u redu. Pri dodavanju novog elementa u listu, taj se pokazivač može ažurirati u O(1) vremenu.

Ove dva jednostavna ATP-a možemo kasnije koristiti za realizaciju algoritama pretrage po grafu. To im nije jedina namjena, pa stog možemo koristiti i za pretvaranje rekurzivnih algoritama u iterativne algoritme.

```
typedef struct _Element {
	int b;
	struct _Element *next;
} Element;

typedef struct {
	Element *head;
	Element *tail;
} Queue;

Queue *create(int size) {
	Queue *q = (Queue*)malloc(sizeof(Queue));
	q->head = q->tail = NULL;
	return q;
}

void enqueue(Queue *q, int b) {
	Element *e = (Element*)malloc(sizeof(Element));
	e->b = b;
	e->next = NULL;
	if (q->tail != NULL)
		q->tail->next = e;
	q->tail = e;
	if (q->head == NULL)
		q->head = e;
}

int dequeue(Queue *q) {
	if (q->head == NULL)
		return -1;
	int e = q->head->b;
	if (q->head == q->tail) {
		free(q->head);
		q->head = q->tail = NULL;
	}
	else {
		Element *f = q->head;
		q->head = q->head->next;
		free(f);
	}
	return e;
}
```

Prioritetni red

Ako u nekakvoj strukturi želimo držati elemente koji imaju prioritet za obradu odnosno vađenje iz reda potreban nam je ATP prioritetni red. Svaki element će uz sadržaj imati broj koji označava prioritet. U nastavku ćemo smatrati da veći broj znači veći prioritet, ali jednostavnom negacijom prioriteta možemo sve okrenuti naopako. Implementacija prioritetnog reda bi se mogla izvesti pomoću nesortiranog ili sortiranog niza ili liste, ali cijena nekih operacija bi bila O(n) što je dosta neefikasno. Na primjer, sa sortiranom listom, vađenje elementa bi bilo relativno brzo, ali bi za dodavanje novog elementa morali šetati kroz listu i dodati ga po redu na odgovarajuće mjesto. Ako bi koristili nesortiranu listu, tada bi dodavanje bilo brzo, ali bi za vađenje elementa morali proći kroz cijelu listu da bi pronašli najveći element. Zbog toga se prioritetni red implementira sa posebnim strukturama ili posebnom logikom. Ovdje će biti prikazana implementacija prioritetnog reda pomoću maksimalne binarne gomile (*max-heap*), gdje će se elementi sastojati samo od prioriteta. Osnovna struktura podataka je običan niz (brojeva), ali logika po kojoj smještamo elemente u taj niz će nam garantirati dobra vremena za dodavanje i vađenje najvećeg elementa. Logički će se elementi slagati u niz tako da formiraju jednu vrstu binarnog stabla. Binarno stablo je hijerarhijska struktura gdje svaki element stabla (roditelj) ima logičku vezu prema 1 ili 2 elementa (djeca). Kasnije ćemo vidjeti i druge vrste stabala, ali za sada ćemo binarno stablo koristiti za logički prikaz maksimalne binarne gomile. Na slici se može vidjeti kako izgleda niz i pripadajući prikaz stabla u nekom trenutku maksimalne gomile.



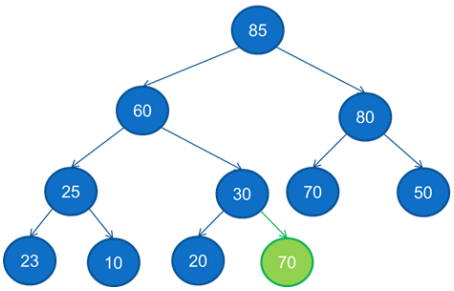
Iz slike se može vidjeti da se iz niza u stablo (i obratno) elementi slažu po razinama, s lijeva na desno, dok god se razina ne ispuni. U prikazu stabla se može vidjeti i osnovno svojstvo binarne maksimalne gomile:

svaki roditelj je veći ili jednak od oba djeteta. Element sa najvećim prioritetom će se uvijek nalaziti na vrhu stabla odnosno na početku niza. Potrebno je definirati dvije operacije dodavanja elementa i uklanjanja maksimalnog elementa tako da gomila i dalje zadrži svojstvo maksimalne gomile. Novi element se uvijek dodaje na kraj niza, odnosno na prvo slobodno mjesto u stablu. Zatim se izvršava procedura popravljanja gomile „od dna“ prema vrhu. To znači da krenuvši od dodanog elementa, uspoređujemo ga sa njegovim roditeljem i ako je veći, mijenjaju mjesto. Zatim se uspoređuje taj roditelj sa svojim roditeljem i ponovno mijenja mjesto ako je veći. Ta procedura se nastavlja sve dok ne nađemo element koji je manji od svog roditelja ili dođemo do vrha (odnosno elementa na indeksu 0). Uklanjanje najvećeg elementa (korijena stabla) se vrši tako da uzmemo element na vrhu, a na njegovo mjesto stavimo zadnji element u stablu. Tada se započinje procedura popravljanja gomile od vrha prema dnu. Krećemo od elementa na vrhu i uspoređujemo ga s oba djeteta. Element (roditelj) mijenja mjesto sa većim djetetom, pod uvjetom da je manji od njega. Ta procedura se nastavlja dok ne dođemo do dna, odnosno pređemo indeks zadnjeg elementa. Da bi se obje procedure izvršavale efikasno na nizu, potrebno je moći brzo odrediti indeks roditelja i oba djeteta za bilo koji element niza. Za nizove gdje indeksi kreću od 0, formule za određivanje roditelja i djece nekog elementa na indeksu *k* su iduću:

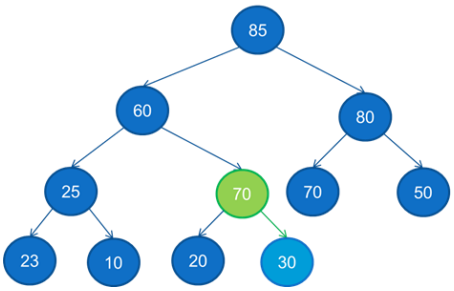
- Roditelj = (k-1) / 2 zaokruženo na nižu vrijednost
- Lijevo dijete = 2k+1
- Desno dijete = 2k+2

Na slici možete vidjeti da sve formule odgovaraju logičkom prikazu elemenata u nizu. Složenost svake operacije je O(logN) gdje je N broj elemenata u gomili. To proizlazi iz činjenice da procedure popravljanja gomile imaju najviše logN koraka što odgovara broju razina u stablu.

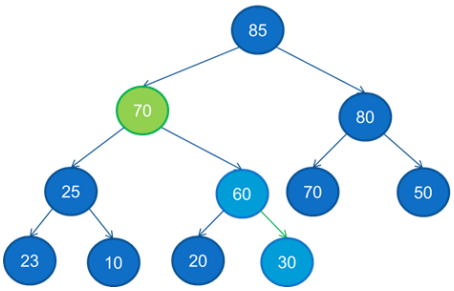
{ 85, 60, 80, 25, 30, 70, 50, 23, 10, 20, 70 }



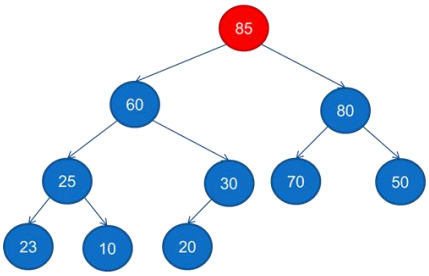
{ 85, 60, 80, 25, 70, 70, 50, 23, 10, 20, 30 }



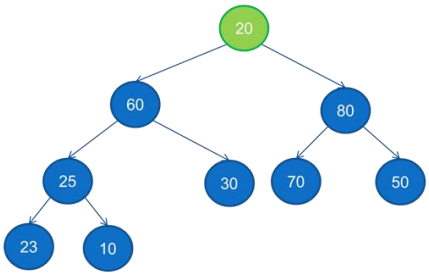
{ 85, 70, 80, 25, 60, 70, 50, 23, 10, 20, 30 }



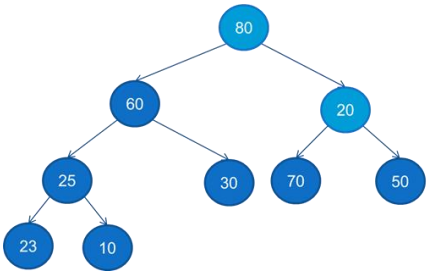
{ 85, 60, 80, 25, 30, 70, 50, 23, 10, 20 }



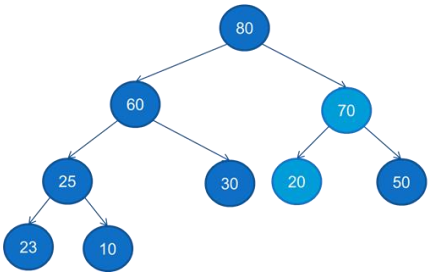
{ 20, 60, 80, 25, 30, 70, 50, 23, 10 }



{ 80, 60, 20, 25, 30, 70, 50, 23, 10 }



{ 80, 60, 70, 25, 30, 20, 50, 23, 10 }



Heapsort

Upotrebom binarne gomile, moguće je realizirati još jedan algoritam sortiranja koji je u istoj klasi složenosti kao quicksort ili mergesort algoritmi – O(NlogN). Algoritam se izvodi u 3 koraka:

1. Pronađemo zadnji element niza koji ima djecu. Taj element se nalazi točno na polovici niza.
2. Pretvaramo niz u binarnu maksimalnu gomilu popravljanjem prema dolje svaki element sa djecom krenuvši od zadnjeg elementa sa djecom prema početku niza (odnosno vrhu stabla).
3. Vadimo sve elemente iz gomile tako da uvijek uzimamo najveći element i popravljamo gomilu, operacija vađenja najvećeg elementa iz gomile. Izvađeni element se jednostavno zamijeni sa zadnjim elementom u skladu sa operacijom vađenja najvećeg elementa iz gomile i koristeći upravo oslobođeno mjesto na kraju niza.

Složenost drugog koraka je O(NlogN), ali u stvarnosti je i bolja jer popravljamo samo polovicu elemenata sa cijenom manjom od O(logN) za popravljanje svakog elementa prema dolje. Složenost trećeg koraka je O(NlogN) jer vadimo N elemenata i svaki put popravljamo gomilu prema dolje sa cijenom O(logN). U konačnici je složenost O(NlogN). Memorijska složenost O(1) jer se sve operacije mogu realizirati bez rekurzije. To je teoretski bolje i od quicksorta i od mergesorta, ali u praksi je quicksort i dalje brži.

Zadaci

Implementirati stog i red pomoću niza i liste. Paziti da sve operacije imaju konstantnu složenost O(1).

Implementirati heapsort algoritam.