

# Implementacija skupa pomoću bit-vektora

- Skup se prikazuje poljem bitova- svakom mogućem elementu skupa odgovara 1 bit, ako je element u skupu njegov bit sadrži vrijednost 1, inače 0
- Uzmimo za primjer skupove čiji su elementi cijeli brojevi od 0 do N
- Veličina polja za prikaz skupa u byte-ovima:  
ako rabimo polje cijelih brojeva: 1 podatak = 4 byte-a = 32 bita,  $M=(N+1)/32$
- Operatori između bitova:
  - & - logički "i": rezultat je 1 ako su oba bita 1
  - | - logički "inkluzivni ili": rezultat je 1 ako je bar jedan bit 1
  - ^ - logički "ekskluzivni ili": rezultat je 1 ako je točno jedan bit 1 (tj. ako su različiti)
  - << i >> - operatori pomaka određenog broja bitova
- Zadatak: napisati program kojim se pomoću bit-vektora prikažu 2 skupa cijelih brojeva čiji elementi se nalaze između 0 i N (određuje se na početku izvođenja programa). Napisati funkcije koje nađu presjek, uniju i razliku ta 2 skupa. Ispisati sve skupove.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef unsigned int ele;
int seleb=sizeof(ele), sele, no_cel;

void nul_skup(ele *S){
    int ind;
    for (ind=0;ind<no_cel;ind++) {
        S[ind]=0;}
}

void puni_skup(ele *S, int noel, ele maxel){
    int ind,no_by,no_bi;
    ele novi;
    ind=0;
    do {
        novi=(ele) maxel *((float) rand()/(RAND_MAX+1));
        no_by=novi/sele;
        no_bi=novi%sele;
        if ((S[no_by] & 1<<no_bi)==0) {
            S[no_by]=S[no_by] | 1<<no_bi;
            ind++; }
    } while(ind<noel);
}

```

```

void ispis_skup(ele *S){
    int ind,ibit;
    printf("Ispis elemenata skupa:\n");
    for (ind=0;ind<no_cel;ind++) {
        for (ibit=0;ibit<sele;ibit++) {
            if (S[ind] & 1<<ibit) printf("%d ",ind*sele+ibit);
        }
        printf("\n"); }
}

```

```

ele * presjek(ele *A, ele *B){
    ele *P;
    int ind;
    P=malloc(seleb*no_cel);
    for (ind=0;ind<no_cel;ind++)
        P[ind]=A[ind] & B[ind];
    return &P[0];
}

```

```

ele * unija(ele *A, ele *B){
    ele *P;
    int ind;
    P=malloc(seleb*no_cel);
    for (ind=0;ind<no_cel;ind++)
        P[ind]=A[ind] | B[ind];
    return &P[0];
}

```

```

ele * razlika(ele *A, ele *B){
    ele *P, *R;
    int ind;
    P=malloc(seleb*no_cel);
    R=malloc(seleb*no_cel);
    for (ind=0;ind<no_cel;ind++) {
        R[ind]=B[ind] ^ 0xffffffff;
        P[ind]=A[ind] & R[ind];}
    return &P[0];
}

```

```

int main(){
    int ind, noa, nob;
    ele maxel, *A, *B, *C, *D, *E,*F;
    sele=seleb*8;
    printf("Unesite vrijednost maksimalnog elementa skupova\n");
    scanf("%d",&maxel);
    no_cel=maxel/(sele)+1;
    A=malloc(seleb*no_cel);
    B=malloc(seleb*no_cel);
    printf("Koliko elemenata ima skup A\n");
    scanf("%d",&noa);
    printf("Koliko elemenata ima skup B\n");
    scanf("%d",&nob);
    nul_skup(A,noa);
    nul_skup(B,nob);
}

```

```
srand(time(NULL));
srand(time(NULL)/rand());
printf("Skup A:\n");
puni_skup(A,noa,maxel);
ispis_skup(A);
printf("Skup B:\n");
puni_skup(B,nob,maxel);
ispis_skup(B);
C=presjek(A,B);
printf("Presjek skupova:\n");
ispis_skup(C);
D=unija(A,B);
printf("Unija skupova:\n");
ispis_skup(D);
E=razlika(A,B);
printf("Razlika A bez B:\n");
ispis_skup(E);
F=razlika(B,A);
printf("Razlika B bez A:\n");
ispis_skup(F);

system("PAUSE");
return 0;
}
```

# Sortiranje pomoću hrpe (heapsort)

- Na prijašnjim vježbama već smo radili dva algoritma sortiranja (sortiranje umetanjem - Insertion Sort i mjehuričasto sortiranje - Bubble Sort) koji su ulazni niz od  $n$  elemenata sortirali u vremenu proporcionalnom s  $n^2$ .
  - Sad ćemo obraditi još jedan algoritam sortiranja – sortiranje pomoću hrpe - Heapsort – koji je efikasniji od Insertion Sorta i Bubble Sorta. Ovaj algoritam polje od  $n$  elemenata sortira u vremenu  $O(n \cdot \log n)$ .
  - Prije nego izložimo algoritam za ovo sortiranje, slijedi najosnovnije o stablastoj strukturi podataka zvanoj **hrpa**.
  - Potpuno binarno stablo  $T$  je hrpa (heap) ako su ispunjeni uvjeti:
    - čvorovi od  $T$  su označeni podacima nekog tipa za koje je definiran totalni uređaj
    - neka je  $i$  bilo koji čvor od  $T$ . Tada je oznaka od  $i$  manja ili jednaka od oznake bilo kojeg djeteta od  $i$  – *minimalna hrpa*
  - također, može biti ovako:
    - neka je  $i$  bilo koji čvor od  $T$ . Tada je oznaka od  $i$  veća ili jednaka od oznake bilo kojeg djeteta od  $i$  – *maksimalna hrpa*
- Općenito: uređaj među podacima može biti i neki drugi, pa se može promijeniti i relacija roditelj - djeca*

- uzimamo da je oznaka roditelja veća ili jednaka od oznaka svih potomaka
- Hrpa je struktura podataka koja se obično implementira pomoću polja
- elementi hrpe spremljeni su u polje koje promatramo kao potpuno binarno stablo (jer je svaki nivo stabla, osim posljednjega, do kraja ispunjen, a čvorovi na posljednjem nivou su "gurnuti" u lijevu stranu)
- Lako je vidjeti da se lijevo dijete čvora  $i$  nalazi na poziciji  $2*i+1$ , a desno na poziciji  $2*i+2$ . Roditelj čvora  $i$  nalazi se na poziciji  $(i-1)/2$
- Za svaki čvor u stablu, definiramo njegovu **visinu** kao broj grana na najdužem putu od tog čvora do nekog lista. **Visinu stabla** definiramo kao visinu njegovog korijena
- Budući da je  $n$ -elementna hrpa poseban slučaj potpunog binarnog stabla, njezina visina je jednaka  $\log_2 n$ . Osnovna operacija za rad s hrpom (Heapify) svoj posao obavlja u vremenu koje je, u najgorem slučaju, proporcionalno visini stabla i stoga se izvršava u vremenu proporcionalnom  $\log_2 n$
- Operacije za rad s hrpom su sljedeće:
  - Funkcija Heapify koja je ključna za očuvanje uređenosti hrpe ( $\log_2 n$ )
  - Funkcija BuildHeap koja od neuređenog ulaznog polja stvara hrpu ( $n*\log_2 n$ )
  - Funkcija HeapSort koja sortira polje ( $n*\log_2 n$ )

## Implementacija strukture podataka HeapType u C-u

- definiramo novi tip podataka, HeapType, kao strukturu koja sadrži:

```
struct heaptype {  
    int * Elements;    //pokazivač na polje elemenata hrpe  
    int Size;         //veličina hrpe  
};  
typedef struct heaptype HeapType;
```

- Funkcija MakeEmptyHeap stvara praznu hrpu zadane veličine, dok funkcija FreeHeap oslobodi memoriju koju je neka hrpa zauzimala:

```
HeapType MakeEmptyHeap(int size) {  
    HeapType heap;  
    heap.Elements = malloc(size * sizeof(int));  
    heap.Size = size;  
    return heap;  
}
```

```
void FreeHeap(HeapType * hptr) {  
    free(hptr->Elements);  
}
```



## Operacija Heapify

- Heapify je operacija koja kao ulazni parametar prima pokazivač na strukturu HeapType i index  $i$  nekog elementa u polju Elements. U času pozivanja funkcije Heapify pretpostavlja se da i lijevo i desno podstablo čvora  $i$  zadovoljavaju svojstvo uređenosti hrpe, ali da je element s indeksom  $i$  (eventualno) manji od svoje djece i time (eventualno) narušava uređenost hrpe
- Funkcija Heapify "pomiče" element upisan u čvor  $i$  prema dolje tako da na kraju podstabla hrpe kojemu je korijen čvor  $i$  postane ispravna podhrpa (tj. podhrpa koja zadovoljava svojstvo uređenosti hrpe)
- Left i Right su jednostavne pomoćne funkcije koje vraćaju indeks lijevog odnosno desnog djeteta čvora  $i$ :

```
int Left(int i) {  
    return 2*i + 1;  
}
```

```
int Right(int i) {  
    return 2*i+2;  
}
```

```

void Heapify(HeapType * hptr, int i) {
    int largest;
    int lft;
    int rht;
    int next = 1;
    do    {
        lft = Left(i);
        rht = Right(i);
        if(lft < hptr->Size && hptr->Elements[lft] > hptr->Elements[i])
            largest = lft;
        else
            largest = i;
        if(rht < hptr->Size && hptr->Elements[rht] > hptr->Elements[largest])
            largest = rht;
        if(largest != i)    {
            int temp = hptr->Elements[i];
            hptr->Elements[i] = hptr->Elements[largest];
            hptr->Elements[largest] = temp;
            i = largest;
        }
        else
            next = 0;
    }
    while(next);
}

```

## Operacija BuildHeap

- Sukcesivnom upotrebom operacije Heapify možemo od "neispravne" hrpe načiniti ispravnu (koja zadovoljava uređenost hrpe). Funkcija izgleda ovako:

```
void BuildHeap(HeapType * heapptr) {  
    for (int i = ParentOfLastElement(heapptr); i >= 0; i--)  
        Heapify(heapptr, i);  
}
```

- BuildHeap koristi pomoćnu funkciju ParentOfLastElement koja vraća indeks roditelja posljednjeg elementa u hrpi. Ta funkcija izgleda ovako:

```
int ParentOfLastElement(HeapType * h) {  
    return Parent(h->Size - 1);  
}
```

```
int Parent(int k) {  
    return (k - 1) / 2;  
}
```

## Algoritam HeapSort

- Algoritam Heapsort najprije pomoću funkcije BuildHeap od ulaznog polja stvori ispravnu hrpu. Nakon toga, stvorenu hrpu pretvara u uzlazno sortirano polje. Funkcija izgleda ovako:

```
void HeapSort(int a[], int n) {
    HeapType heap;
    heap.Elements = a;
    heap.Size = n;
    BuildHeap(&heap);
    for(int i = n - 1; i > 0; i--)
    {
        int temp = heap.Elements[i];
        heap.Elements[i] = heap.Elements[0];
        heap.Elements[0] = temp;
        heap.Size--;
        Heapify(&heap, 0);
    }
}
```

- Vidimo da algoritam najprije poziva BuildHeap (što traje proporcionalno s  $n \cdot \log_2 n$ ) a zatim u osnovi "vrti petlju" u kojoj se  $n-1$  puta obavljaju neke jednostavne operacije i poziva funkcija Heapify koja traje proporcionalno s  $\log_2 n$ . Sveukupno, stoga, funkcija HeapSort se izvršava u vremenu proporcionalnim s  $n \cdot \log_2 n$ , što je bitno bolje od sortiranja umetanjem (Insertion Sort) ili mjehuričastog sortiranja (Bubble Sort) čije je vrijeme izvršavanja dano s  $n^2$ .
- Kompletan kod za sortiranje pomoću hrpe: primjer
- Napisati program koji od ulaznog polja cijelih brojeva izgradi hrpu, te ispiše polje u kojem su spremljeni elementi hrpe. Također napraviti sortiranje polja cijelih brojeva upotrebom sortiranja pomoću hrpe.

```
#include <stdio.h>
#include <stdlib.h>
#define MaxSize 20

struct heapType {
    int * Elements;
    int Size;
};

typedef struct heapType HeapType;

int Left(int k) {
    return 2*k + 1;
}

int Right(int k) {
    return 2*k + 2;
}

int Parent(int k) {
    return (k - 1) / 2;
}

int ParentOfLastElement(HeapType * h) {
    return Parent(h->Size - 1);
}
```

```

HeapType MakeEmptyHeap(int size) {
    HeapType heap;
    heap.Elements = malloc(size * sizeof(int));
    heap.Size = size;
    return heap;
}

void FreeHeap(HeapType * heapptr) {
    free(heapptr->Elements);
}

void Heapify(HeapType * heapptr, int i) {
    int largest, lft, rht, dalje = 1;
    do {
        lft = Left(i);      rht = Right(i);
        if(lft < heapptr->Size && heapptr->Elements[lft] > heapptr->Elements[i])
            largest = lft;
        else largest = i;
        if(rht < heapptr->Size && heapptr->Elements[rht] > heapptr->Elements[largest])
            largest = rht;
        if(largest != i) {
            int temp = heapptr->Elements[i];
            heapptr->Elements[i] = heapptr->Elements[largest];
            heapptr->Elements[largest] = temp;
            i = largest;      } else
            dalje = 0; } while(dalje);
}

```

```

void BuildHeap(HeapType * heapptr)
{
    for(int i = ParentOfLastElement(heapptr); i >= 0; i--)
        Heapify(heapptr, i);
}

void HeapSort(int a[], int n)
{
    HeapType heap;
    heap.Elements = a;
    heap.Size = n;
    BuildHeap(&heap);
    for(int i = n - 1; i > 0; i--)
    { // u korijenu je najveći element hrpe
        int temp = heap.Elements[i];
        heap.Elements[i] = heap.Elements[0]; // korijen stavljamo na kraj polja
        heap.Elements[0] = temp; // zadnji element polja ide u korijen
        heap.Size--; // smanjimo hrpu za 1
        Heapify(&heap, 0); // slozi se nova hrpa s n-1 čvorova
    }
}

```



```

int main() {
    int i, a[MaxSize] = {4,0,11,1,21,-3,26,2,5,16,17,9,10,13,14,25,8,3,7,-9};
    HeapType myheap = MakeEmptyHeap(MaxSize);

    for(i = 0; i < myheap.Size; i++)
        myheap.Elements[i] = a[i];
    printf("Ulazno polje:\n");
    for(i = 0; i < myheap.Size; i++)
        printf("%i ", myheap.Elements[i]);          printf("\n\n");
    BuildHeap(&myheap);
    printf("Nakon kreiranja hrpe:\n");
    for(i = 0; i < myheap.Size; i++)
        printf("%i ", myheap.Elements[i]);          printf("\n\n\n");

    FreeHeap(&myheap);
    printf("Prije sortiranja:\n");
    int b[MaxSize] = {4,0,11,1,21,-3,26,2,5,16,17,9,10,13,14,25,8,3,7,-9};
    for(i = 0; i < MaxSize; i++)
        printf("%i ", b[i]);  printf("\n\n");
    HeapSort(b, MaxSize);
    printf("Nakon sortiranja:\n");
    for(i = 0; i < MaxSize; i++)
        printf("%i ", b[i]);  printf("\n");
    system("PAUSE");
    return 0;
}

```