

## 6. Data Structures [WiP]

---

José Proença

Algorithms (CC4010) 2023/2024

CISTER – U.Porto, Porto, Portugal

<https://cister-labs.github.io/alg2324>



**CISTER** - Research Centre in  
Real-Time & Embedded  
Computing Systems

- Stacks/Queues/PriorityQueues
- Hashtables/Search trees
- Graphs
  - Depth/Breathfirst traversals
  - Acyclic – topological order
  - Transitive closure
  - Minimum spanning tree
  - Shortest/longest path

## We have seen that

Different **data structures** are better at different **operations**

## We will see

Useful data structures and associated operations (code)

## Examples

Arrays can have operations to implement sets, multisets, trees, etc.

# Sets and Sequences

---

# Sets and Multisets

```
#define MAXS 100  
typedef char SetInt [MAXS] ;
```

Given SetInt s:

$5 \in s \Leftrightarrow s[5] \neq 0$

```
#define MAXMS 100  
typedef int MSetInt [MAXS] ;
```

Given MSetInt ms:

$\{4, 4\} \subseteq ms \Leftrightarrow ms[4] \leq 2$

# Sets and Multisets – operations

```
void initSet      (SetInt);  
int  searchSet   (SetInt, int);  
int  addSet      (SetInt, int);  
int  emptySet    (SetInt);  
void unionSet    (SetInt, SetInt,  
                  SetInt);  
void intersectSet (SetInt, SetInt,  
                  SetInt);  
void differenceSet (SetInt, SetInt,  
                   SetInt);
```

```
void initMSet     (MSetInt);  
int  searchMSet   (MSetInt, int);  
int  addMSet      (MSetInt, int);  
int  emptyMSet    (MSetInt);  
void unionMSet    (MSetInt, MSetInt,  
                  SetInt);  
void intersectMSet (MSetInt, MSetInt,  
                  MSetInt);  
void differenceMSet (MSetInt, MSetInt,  
                   MSetInt);
```

**Ex. 6.1:** What is the expected cost of each function? Could you implement them?

# Sequences – Recall linked lists

```
typedef struct list { int value ;  
    struct list *next;  
} *LInt;
```

```
LInt add (int x, LInt l) {  
    LInt new =  
        malloc(sizeof(struct list));  
    if (new != NULL) {  
        new->value=x;  
        new->next=l ;  
    }  
    return new;  
}
```

```
LInt dda (int x, LInt l) {  
    LInt pt = l;  
    while (pt != NULL) pt = pt->next;  
    pt = malloc(sizeof(struct list));  
    pt -> next = x;  
    pt -> next = NULL ;  
    return l ;  
}
```

## Sequences – Recall linked lists (fixed)

```
typedef struct list { int value ;  
    struct list *next;  
} *LInt;
```

```
LInt add (int x, LInt l) {  
    LInt new = malloc(sizeof(struct  
        list ));  
    if (new != NULL) {  
        new->value=x;  
        new->next=l ;  
    }  
    return new;  
}
```

```
LInt dda (int x, LInt l) {  
    LInt pt = l, prev;  
    while (pt != NULL) {  
        prev = pt; pt = pt->next; }  
    pt = malloc(sizeof(struct list));  
    pt->next = x;  
    pt->next = NULL ;  
    if (l==NULL) l = pt;  
    else prev->prox = pt;  
    return l;  
}
```

**Ex. 6.2:** What is the possible complexity of lookup, concat, reverse?



# Sequences – reverse analysis

```
LInt reverse1 (LInt l) {  
    LInt r, pt;  
    if (l==NULL || l->next==NULL)  
        r=l;  
    else {  
        r = pt = reverse1 (l->next);  
        while (pt->next != NULL)  
            pt = pt->next;  
        pt->next = l;  
        l->next = NULL;  
    }  
    return r; }
```

```
LInt reverse2 (LInt l) {  
    LInt r, tmp;  
    r = NULL;  
    while (l !=NULL) {  
        tmp=l; l=l->next;  
        tmp->next=r; r=tmp;  
    }  
    return r;  
}
```

**Ex. 6.3:** What is the complexity of each reverse?

**Ex. 6.4:** What is the (informal) loop invariant in reverse2, assuming:  
pre:  $l == l_0$  and post:  $r == \text{rev}(l_0)$ ?

```
https://docs.scala-lang.org/  
overviews/collections-2.13/  
performance-characteristics.html
```

## Buffers (stacks and queueus)

---

```
#define MAX 1000
typedef struct stack {
    int values [MAX];
    int sp;
} Stack;
```

```
typedef struct cell {
    int value;
    struct cell *next;
} Cell , *Stack;
```

```
typedef struct stack {
    int size;
    int *values;
    int sp;
} Stack;
```

```
#define MAX 1000
typedef struct stack {
    int values [MAX];
    int sp;
} Stack;
```

with static arrays

```
typedef struct cell {
    int value;
    struct cell *next;
} Cell , *Stack;
```

with linked lists

```
typedef struct stack {
    int size;
    int *values;
    int sp;
} Stack;
```

with dynamic arrays

**Ex. 6.5:** (Informally) what is the expected complexity of: push, pop, head?

## Exercise: Push-pop with dynamic arrays

```
void push (Stack *s , int x){
    if (s->sp == s->size)
        doubleArray (s);
    if (r == 0)
        s->values[s->sp++] = x;
}

void doubleArray (Stack *s){
    s->size *= 2;
    s->values =
        realloc(s->values , s->size);
}
```

```
int pop (Stack *s){
    // reduces by half when only
    // 25% capacity is used
    ...
}

void halfArray (Stack *s){
    ...
}
```

**Ex. 6.6:** Implement the optimised pop function and discuss its complexity.

```
#define MAX 1000
typedef struct queue
{
    int values [MAX];
    int start, size;
} Queue;
```

```
typedef struct cell {
    int value ;
    struct cell *prox ;
} Cell ;

typedef struct queue {
    struct cell *start, *end;
} Queue;
```

```
typedef struct queue
{
    int max;
    int *values;
    int start, size;
} Queue;
```

# Queues

```
#define MAX 1000
typedef struct queue
{
    int values [MAX];
    int start, size;
} Queue;
```

with static arrays  
(circular)

```
typedef struct cell {
    int value ;
    struct cell *prox ;
} Cell ;

typedef struct queue {
    struct cell *start, *end;
} Queue;
```

with linked lists

```
typedef struct queue
{
    int max;
    int *values;
    int start, size;
} Queue;
```

with dynamic arrays  
(circular)

**Ex. 6.7:** (Informally) what is the complexity of: `init`, `isEmpty`, `enqueue`, `dequeue`?



- Binary tree
- Each node is larger than any of its children
- Implemented as an array

```
#define MAX 1000
typedef struct prQueue {
    int values [MAX];
    int size ;
} PriorityQ ;
```

## Tree example in the board

```
size=17    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
values:    [10 15 11 16 22 35 20 21 23 34 37 80 43 22 25 24 28]
```

**Ex. 6.8:** Using the previous example, provide an expression to:

1. calculate the index of the *left* tree given a position  $i$
2. calculate the index of the *right* tree given a position  $i$
3. calculate the index of the *parent* of a given a position  $i$
4. calculate the index of the index of the *first leaf*

**Ex. 6.9:** Define `bubbleUp(int i, int h[])`

Fixes a min-heap by swapping the  $i$ -th element with the parent while needed.

**Ex. 6.10:** Define `bubbleDown(int i, int h[], int N)`

Fixes a min-heap by swapping the  $i$ -th element with one of the children while needed.

## Ex. 6.11: Define the following operations:

- `void empty (PriorityQueue *q)` – initialises the queue;
- `int isEmpty (PriorityQueue *q)` – tests if `q` is empty;
- `int add (int x, PriorityQueue *q)` – adds a value `x`, returning 0 when the queue is full;
- `int remove (PriorityQueue *q, int *rem)` – removes the next element, and copies it to *rem*.

# Dictionaries

---

Dictionary: maps keys to values

(Keys are unique)

## Idea

- *Magic function* `hash` converts a key into an `index` (number).
- This `index` points to the position of an array where the value *should* be found.
- Usually the size of the array is `less` than the set of possible keys, i.e., `hash` is not injective.
- If 2 keys have the same `hash` value, there is a `colision` that must be mitigated (alternative solutions exist).

# Hashtables: Closed and Open Addressing

## Closed Addressing (or chaining)

- Table = *array of linked lists*
- Find value of key  $k$ :
  - go to index  $\text{hash}(k)$
  - traverse list until  $k$

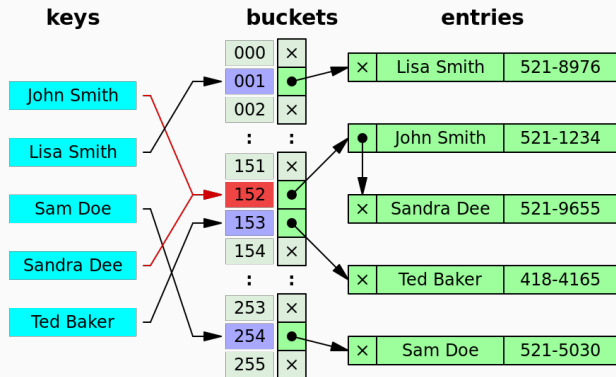
## Open Addressing

- Table = *just an array*
- Find value of key  $k$ :
  - go to index  $\text{hash}(k)$
  - “*jump*” until  $k$

## Some concerns

- Use dynamic arrays (grow when the **load factor** ( $\# \text{keys} / \text{HSIZE}$ ) gets high)
  - Need to *rehash*
- Smart *jumps* (probe function to know where to jump)
- Need to *garbage collect* in open addressing

# Intuition: Hashtables with Closed Addressing



(from Wikipedia)

# Hashtables with Closed Addressing

- `int hash(int k, int size);`
- `void initTab(HTChain h);`
- `int lookup(HTChain h, int k, int *i);`
- `int update(HTChain h, int k, int i);`
- `int remove(HTChain h, int k);`

```
#define HSIZE 1000

typedef struct bucket {
    int key;
    int info;
    struct bucket *next;
} *Bucket;

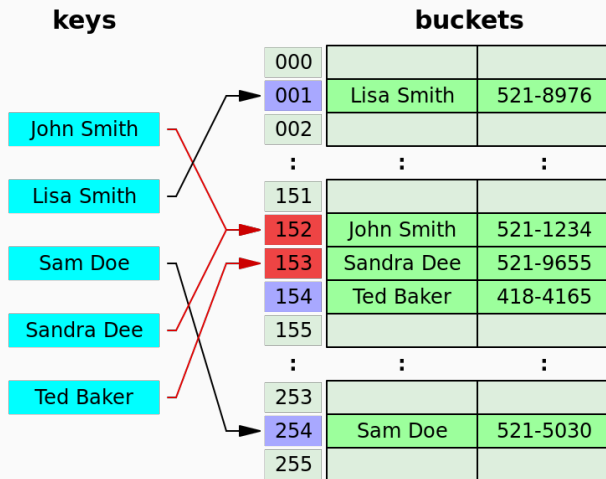
typedef Bucket
    HTChain[HSIZE];
```

**Ex. 6.12:** Implement lookup

**Ex. 6.13:** (Informally) what is the expected complexity of each function?



# Intuition: Hashtables with Open Addressing



(from Wikipedia)

# Hashtables with Open Addressing

- `int hash(int k, int size);`
- `void initTab(HashTable h);`
- `void lookup(HashTable h, int k, int *i);`
- `void update(HashTable h, int k, int i);`
- `void remove(HashTable h, int k);`
- `int find_probe (HashTable h, int k)`
  - linear vs. quadratic probing (why quadratic?)

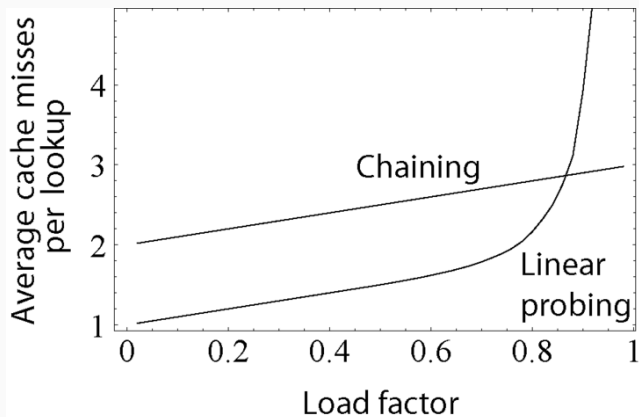
```
#define HSIZE 1000
#define STATUSFREE 0
#define STATUSUSED 1

typedef struct bucket {
    int status ;
    int key;
    int info;
} Bucket ;

typedef Bucket
    HashTable [HSIZE];
```

**Ex. 6.14:** Define a linear probing function and update.

## Lookups: Open vs. Closed



(from Wikipedia)

# Removing with Open Addressing

- `int hash(int k, int size);`
- `void initTab(HashTable h);`
- `void lookup(HashTable h, int k, int *i);`
- `void update(HashTable h, int k, int i);`
- `int find_probe (HashTable h, int k);`
- `void remove(HashTable h, int k);`

```
#define HSIZE 1000
#define STATUSFREE 0
#define STATUSUSED 1
#define STATUSDEL 2

typedef struct bucket {
    int status ;
    int key;
    int info;
} Bucket ;

typedef Bucket
    HashTable [HSIZE];
```

**Ex. 6.15:** How would you implement update?

How would you implement a *garbageCollect* that removes deleted cells?

What is their complexity?

### We will see:

- Height- and weight-balanced tree
- Self-balancing binary search tree
  - AVL tree
  - Red-black tree

## Height-balanced

- more used
- AVL: left-height = right-height  $\pm 1$
- Red-black: similar wrt *black*
- height =  $\log n$

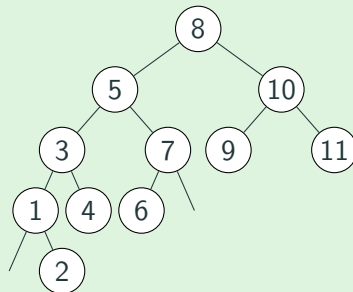
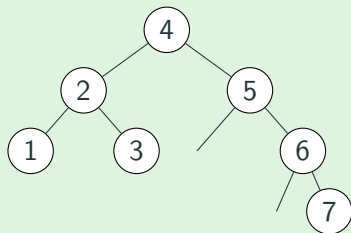
## Weight-balanced

- less used
- leafs-left/right  $\geq \alpha \times \text{leafs}$ ,  $0 < \alpha < 1$
- better for lookup intensive systems

- By Adelson-Velsky and Landis
- Oldest self-balancing binary search tree data structure to be invented ('62)
- Binary (left-right) search (sorted) tree
- Labels in the nodes
- At every node, the height of left and right trees differ at most by 1
- Insertions and removals preserve this

Function	Amortized	Worst Case	<i>Amortized (RB)</i>	<i>Worst case (RB)</i>
Search	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Insert	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Delete	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

## Ex. 6.16: Are these balanced?





# Update in an AVL tree

## See animation

[https://en.wikipedia.org/wiki/AVL\\_tree#/media/File:AVL\\_Tree\\_Example.gif](https://en.wikipedia.org/wiki/AVL_tree#/media/File:AVL_Tree_Example.gif)

4 rotations: left, right, right-left, right-right

```
typedef struct avl {
    int bal;
    int key, info;
    struct avl *left , *right ;
} *AVL;

#define LEFT -1
#define RIGHT 1
#define BAL 0

// returns 0 if key already existed
int updateAVL (AVL *a, int k, int i);
```

**Ex. 6.17:** How would you implement an update without balancing?

**Ex. 6.18:** How would you implement AVL rotateRight(AVL a)?

## Full code: updateAVL

```
int updateAVL(AVL *a,
              int k, int i){
    int g, u;
    *a =
        updateAVLRec(*a,k,i,&g,&u);
    return u;
}
```

```
AVL updateAVLRec(AVL a , int k ,
                 int i, int *g , int *u){
    if (a == NULL) {           // insert k->i here
        a = malloc (sizeof (struct avl ));
        a->key=k; a->info=i ; a->bal=BAL;
        a->left=a->right=NULL; *g=1; *u=0;
    } else if (a->key==k) {     // update k->i
        a->info=i; *g=0; *u=1;
    } else if (a->key > k) {    // update left
        a->left = updateAVLRec(a->left,k,i,g,u);
        if (*g == 1) switch (a->bal){ // balance
            case LEFT: a= fixLeft(a); *g=0; break;
            case RIGHT:a->bal=BAL;      *g=0; break;
            case BAL:  a->bal=LEFT;      break;
        }
    } else{ // a->key < k           update right
        // left <--> right
    }
    return a ;
}
```

## Full code: updateAVL – fix

```
AVL fixLeft(AVL a) {
    AVL b, c;
    b=a->left ;
    if (b->bal==LEFT){
        a->bal = b->bal = BAL;
        a=rotateRight(a);
    } else {
        c = b->right ;
        switch (c->bal) {
            case LEFT:  a->bal=RIGHT; b->bal=BAL;  break;
            case RIGHT: a->bal=BAL;   b->bal=LEFT; break;
            case BAL:   a->bal=BAL;   b->bal=BAL;
        }
        c->bal = BAL;
        a->left = rotateLeft(b);
        a = rotateRight(a);
    }
    return a;
}
```

...

# Graphs

---

- Depth/Breathfirst traversals
- Acyclic – topological order
- Transitive closure
- Minimum spanning tree
- Shortest/longest path