Data Structures – LECTURE 6

Dynamic data structures

- Motivation
- Common dynamic ADTs
- Stacks, queues, lists: abstract description
- Array implementation of stacks and queues
- Linked lists
- Rooted trees
- Implementations

Motivation (1)

- So far, we have dealt with one type of data structure: an array. Its **length** does not change, so it is a **static** data structure. This either requires knowing the length ahead of time or waste space.
- In many cases, we would like to have a **dynamic** data structure whose length changes according to computational needs
- For this, we need a scheme that allows us to store elements in physically different order.

2	0	5	3	4	1

3	2	4	1	0	5

Motivation (2)

- Examples of operations:
  - **Insert**(*S*, *k*): Insert a new element
  - **Delete**(*S*, *k*): Delete an existing element
  - **Min**(*S*), **Max**(*S*): Find the element with the maximum/minimum value
  - **Successor**(*S*,*x*), **Predecessor**(*S*,*x*): Find the next/previous element
- At least one of these operations is usually expensive (takes $O(n)$ time). Can we do better?

Abstract Data Types –ADT

- An abstract data type is a collection of formal specifications of data-storing entities with a well designed set of operations.
- The set of operations defined with the ADT specification are the operations it "supports".
- What is the difference between a data structure (or a class of objects) and an ADT?

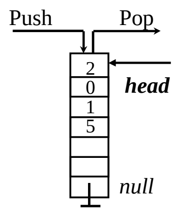
  → The data structure or class is an *implementation* of the ADT to be run on a specific computer and operating system. Think of it as an abstract JAVA class. The course emphasis is on ADTs.

Common dynamic ADTs

- Stacks, queues, and lists
- Nodes and pointers
- Linked lists
- Trees: rooted trees, binary search trees, red-black trees, AVL-trees, etc.
- Heaps and priority queues
- Hash tables

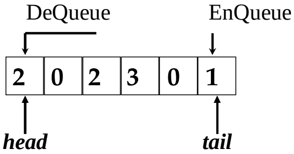
Stacks -- מחסנית

- A stack *S* is a linear sequence of elements to which elements *x* can only be inserted and deleted from the head of the list in the order they appear.
- A stack implements the Last-In-First-Out (LIFO) policy.
- The stack operations are:
  - Stack-Empty(*S*)
  - Pop(*S*)
  - Push(*S*,*x*)



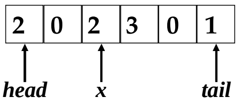
Queues -- תור

- A queue Q is a linear sequence of elements to which elements are inserted at the end and deleted from the beginning.
- A queue implements the First-In-First-Out (FIFO) policy.
- The queue operations are:
  - Queue-Empty(Q)
  - EnQueue(Q, x)
  - DeQueue(Q)



Lists -- רשימות

- A list L is a linear sequence of elements.
- The first element of the list is the head and the last is the tail. When both are null, the list is empty
- Each element has a predecessor and a successor
- The list operations are:
  - Successor(L,x), Predecessor(L,x)
  - List-Insert(L,x)
  - List-Delete(L,x)
  - List-Search(L,k)



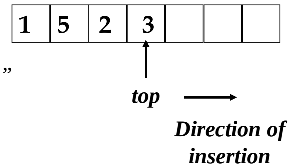
Implementing stacks and queues

- <u>Array implementation</u>
  - use an array A of n elements $A[i]$, where n is the maximum number of elements expected.
  - Top(A), Head(A), and Tail(A) are array indices
  - Stack and queue operations involve index manipulation
  - Lists are not efficiently implemented with arrays
- <u>Linked list</u>
  - Create objects for elements as they appear
  - Do not have to know the maximum size in advance
  - Manipulate pointers

Stacks: array implementation

<u>**Push**</u>(*S*, *x*)

1. **if** *top*[*S*] = *length*[S]
2. **then error** *"overflow"*
3. *top*[*S*] ← *top*[*S*] + 1
4. *S*[*top*[*S*]] ← *x*



<u>**Pop**</u>(*S*)

1. **if** *top*[*S*] = 0
2. **then error** *"underflow"*
3. **else** *top*[*S*] ← *top*[*S*] – 1
4. **return** *S*[*top*[*S*] +1]

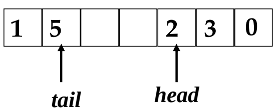
<u>**Stack-Empty**</u>(*S*)

1. **if** *top*[*S*] = 0
2. **then return** *true*
3. **else return** *false*

Queues: array implementation

<u>**Dequeue**</u>(*Q*)

1. *x* ← *Q*[*head*[*Q*]]
2. **if** *head*[*Q*] = *length*[*Q*]
3. **then** *head*[*Q*] ← 1
4. **else** *head*[*Q*] ← $(head[Q]+1)_{\text{mod } n}$
5. **return** *x*



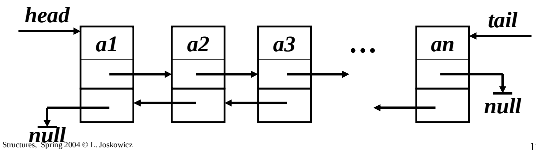
<u>**Enqueue**</u>(*Q*, *x*)

1. *Q*[*tail*[*Q*]] ← *x*
2. **if** *tail*[*Q*] = *length*[*Q*]
3. **then** *tail*[*Q*] ← *x*
4. **else** *tail*[*Q*] ← $(tail[Q]+1)_{\text{mod } n}$

Boundary conditions omitted

Linked Lists -- רשימות מקושרות

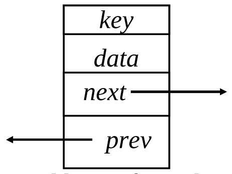
- The physical and logical order of elements need not be the same; instead, use pointers to indicate where the next (previous) element is.
- By manipulating the pointers, we can insert and delete elements without having to move all the others! Lists can be signly or doubly linked.



Nodes and pointers

- A node is an object that holds the data, a pointer to the next node and (optionally), a pointer to the next node. If there is no next node, the pointer is to "null"

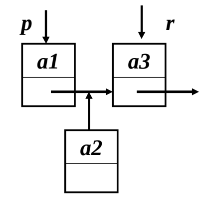
```
Class ListNode {
   Object key;
   Object data;
   ListNode next;
   ListNode prev;
}
```

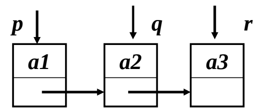


- A pointer indicates the <u>memory address</u> of a node
- Nodes usually occupy constant space: $\Theta(1)$

Example: Insertion

Insertion of a new node *q* between successive nodes *p* and *r*:



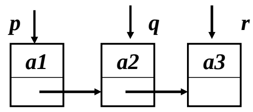


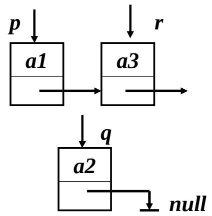
next[*q*] ← *r*

next[*p*] ← *q*

Example: Deletion

Deletion of a node *q* between previous node *p* and successor node *r*





next[*p*] ← *r*

next[*q*] ← *null*

Linked lists operations

<u>**List-Search**</u>(*L*, *k*)

1. *x* ← *head*[*L*]
2. **while** *x* ≠ *null and key*[*x*] ≠ *k*
3. **do** *x* ← *next*[*x*]
4. **return** *x*

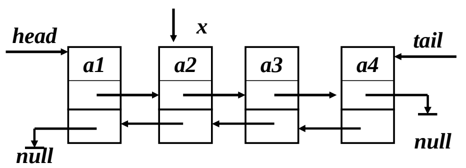
<u>**List-Insert**</u>(*L*, *x*)

1. *next*[*x*] ← *head*[*L*]
2. **if** *head*[*L*] ≠ *null*
3. **then** *prev*[*head*[*L*]] ← *x*
4. *head*[*L*] ← *x*
5. *prev*[*x*] ← *null*

<u>**List-Delete**</u>(*L*, *x*)

1. **if** *prev*[*L*] ≠ *null*
2. **then** *next*[*prev*[*x*]] ← *next*[*x*]
3. **else** *head*[*L*] ← *next*[*x*]
4. **if** *next*[*L*] ≠ *null*
5. **then** *prev*[*next*[*x*]] ← *prev*[*x*]

Example: linked list operations

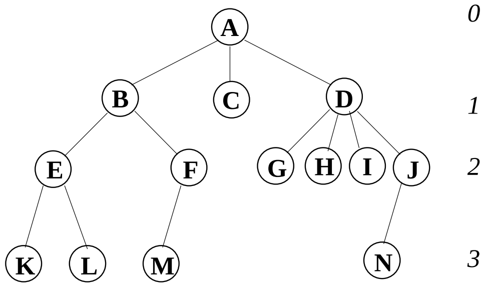


Circular lists: connect first and last elements!

Rooted trees

- A rooted tree **T** is an ADT in which elements are ordered in a tree-like structure.
- A tree consists of nodes, which hold elements, and edges, which show relations between two nodes.
- There are three types of nodes: a root, internal nodes, leaf
- The tree structure is:
  - Connected: there is an edge path from the root to any other node
  - No cycles: there is <u>only one</u> path from the root to a node
  - Each node except the root has a single ancestor
  - Leaves have no outgoing edges
  - Internal nodes have one or more out-going edges (= 2 → binary)

Rooted tree: example



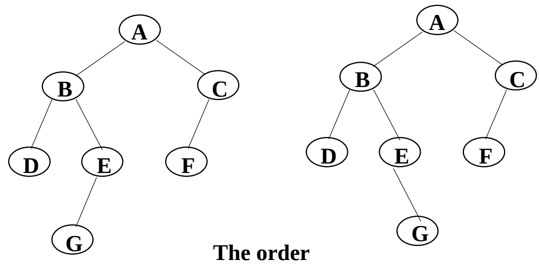
Trees terminology

- Internal nodes have a **parent** and one or more **children**.
- Nodes on the same level are **siblings** (children of the same parent)
- **Ancestor/descendent** relationships – recursive definition of parent and children.
- **Degree of a node**: number of children
- **Path**: a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is a parent of n_{i+1}. The path length is k.
- **Tree height**: length of the longest path from a root to a leaf.
- **Node depth**: length of the path from the root to the node.

Binary trees

- A binary tree **T** is a tree whose root and internal nodes have at most two children.
- Recursively: a binary tree is a tree that either contains no nodes or consists of a root node, and two sub-trees (left and right) each of which is also a binary tree.

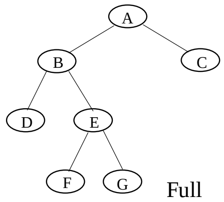
Binary tree: example



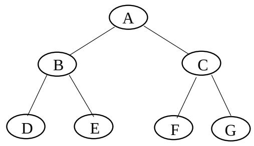
The order matters!!

Full and complete trees

- Full binary tree: each node has either degree 0 (a leaf) or 2 exactly two non-empty children.
- Complete binary tree: a full binary tree in which all leaves have the same depth.



Full



Complete

Properties of binary trees

- How many leaf nodes does a complete binary tree of height d have?

$$2^d$$

- What is the number of internal nodes in such a tree?

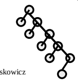
$$1+2+4+\ldots+2^{d-1} = 2^d - 1 \text{ (\textit{less than half!})}$$

- What is the total number of nodes?

$$1+2+4+\ldots+2^{d} = 2^{d+1} - 1$$

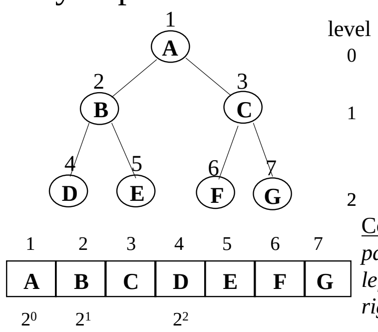
- How tall can a full/complete binary tree with n leaf nodes be?

$(n-1)/2$



$2^{d+1} - 1 = n \rightarrow \log(n+1) - 1 \leq \log(n)$

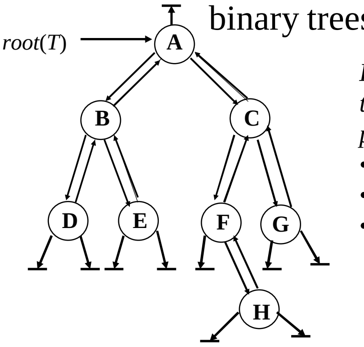
Array implementation of binary trees



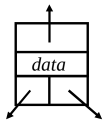
2^d elements at level d

Complete tree:
parent(i) = floor(i/2)
left-child(i) = 2i
right-child(i) = 2i +1

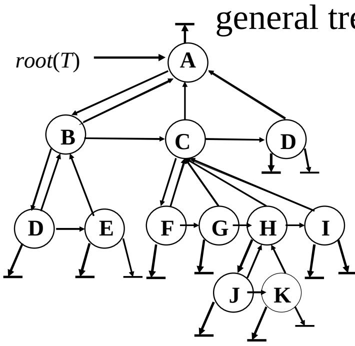
Linked list implementation of binary trees



Each node contains the data and three pointers:
- *ancestor(node)*
- *left-child(node)*
- *right-child(node)*



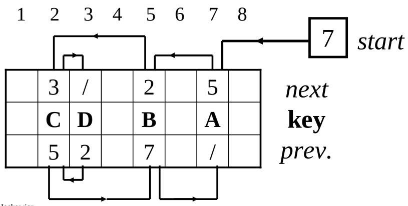
Linked list implementation of general trees



Left-child/right sibling representation
Each node contains the data and three pointers:
- *ancestor*(*node*)
- *left-child*(*node*)
- *right-sibling*(*node*)

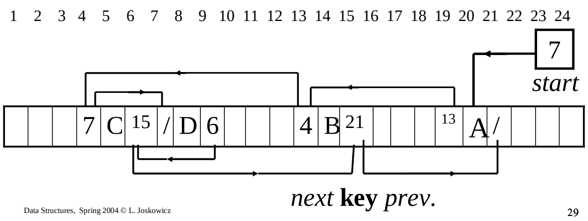
Implementation of pointers and objects

Multiple-array representation of objects:
Each object with k fields is represented as an array with k elements + 2 fields: *previous* and *next*;



Implementation of pointers and objects

Single-array representation of objects:
Each object with k fields is represented with k consecutive fileds 2 fields: *previous* and *next*;



Memory management

- Most languages have a mechanism for allocating and freeing storage objects.
- Memory can thought of as containing two zones: *free memory* and *used memory*.
- Allocating objects: when a new object structure is created, the next available free memory block is used
- De-allocating objects: an object becomes unused when it cannot be reached anymore. Accumulating unused objects is bad since the system can run out of memory unexpectedly.

Allocating and freeing objects

- Two ways to deal with unused objects:
  - the user explicitly frees (de-allocate) objects
  - the system performs "garbage collection" upon request or automatically, once in a while
- When a program terminates, its storage must be recovered (marked free) for otherwise the memory will quickly fill up.
- Keep free objects in a singly linked list managed as a stack → freeing and releasing an object takes $O(1)$.

Code for allocate and free

Allocate-Object()

1. **if** *free = null*
2. **then error** *"out of space"*
3. **else** *x* ← *free*
4. *free* ← *next*[*x*]
5. **return** *x*

Free-Object(*x*)

1. *next*[*x*] ← *free*
2. *free* ← *x*