DATA STRUCTURES AND ALGORITHMS
LECTURE 4

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 3...

- Iterator

- Binary Heap

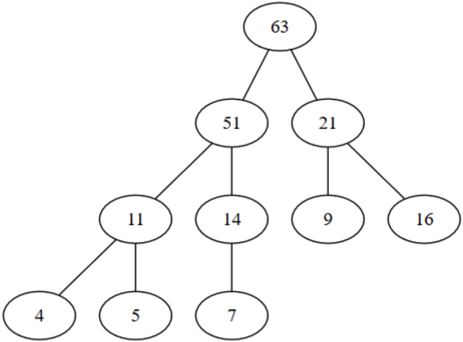
Today

1. Binary Heap

2. Linked Lists
   - Singly Linked Lists

Heap - Add - example

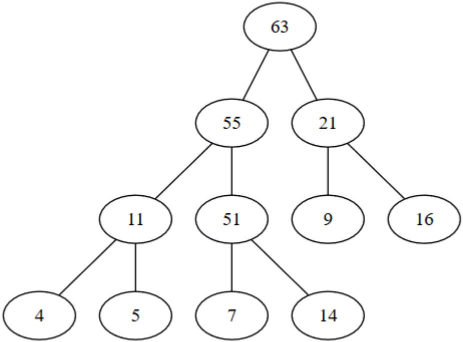
- Consider the following (MAX) heap. Let's add number 55 to it.



63	51	21	11	14	9	16	4	5	7	

Heap - Remove - example

- From a heap we can only remove the root element.



63	55	21	11	51	9	16	4	5	7	14

Heap-sort

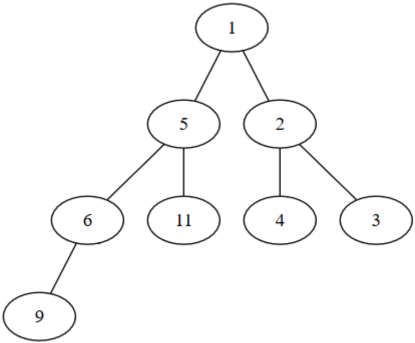
- There is a sorting algorithm, called *Heap-sort*, that is based on the use of a heap.
- In the following we are going to assume that we want to sort a sequence in ascending order.
- Let's sort the following sequence: [6, 1, 3, 9, 11, 4, 2, 5]

Heap-sort - Naive approach

- Based on what we know so far, we can guess how heap-sort works:
    - Build a min-heap adding elements one-by-one to it.
    - Start removing elements from the min-heap: they will be removed in the sorted order.

Heap-sort - Naive approach

- The heap when all the elements were added:



- When we remove the elements one-by-one we will have: 1, 2, 3, 4, 5, 6, 9, 11.

Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?

Heap-sort - Naive approach

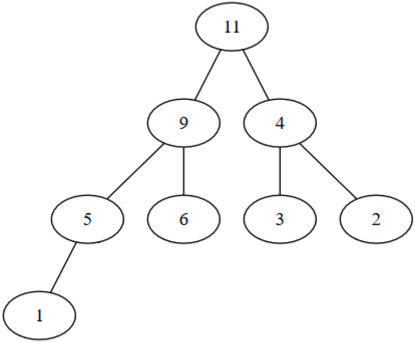
- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2 n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?

Heap-sort - Naive approach

- What is the time complexity of the heap-sort algorithm described above?
- The time complexity of the algorithm is $O(nlog_2n)$
- What is the extra space complexity of the heap-sort algorithm described above (do we need an extra array)?
- The extra space complexity of the algorithm is $\Theta(n)$ - we need an extra array.

Heap-sort - Better approach

- If instead of building a min-heap, we build a max-heap (even if we want to do ascending sorting), we do not need the extra array.



Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.

Heap-sort - Better approach

- We can improve the time complexity of building the heap as well.
  - If we have an unsorted array, we can transform it easier into a heap: the second half of the array will contain leaves, they can be left where they are.
  - Starting from the first non-leaf element (and going towards the beginning of the array), we will just call *bubble-down* for every element.
  - Time complexity of this approach: $O(n)$ (but removing the elements from the heap is still $O(nlog_2 n)$)

Linked Lists

- A *linked list* is a linear data structure, but the order of the elements is determined not by indexes, but by a pointer which is placed in each element.

- A linked list is a structure that consists of *nodes* (sometimes called *links*) and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).

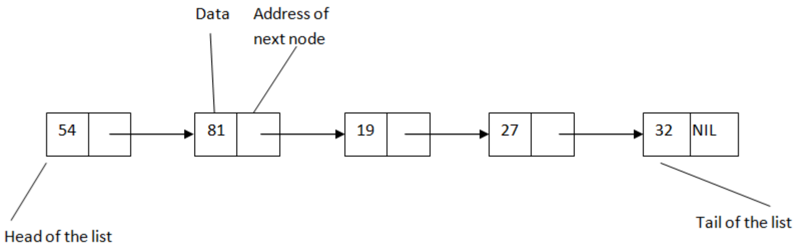
- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.

Linked Lists

- Elements from a linked list are accessed based on the pointers stored in the nodes.
- We can directly access only the first element (and maybe the last one) of the list.

Linked Lists

- Example of a linked list with 5 nodes:



Singly Linked Lists - SLL

- The linked list from the previous slide is actually a *singly linked list - SLL*.
- In a SLL each node from the list contains the data and the address of the next node.
- The first node of the list is called *head* of the list and the last node is called *tail* of the list.
- The tail of the list contains the special value *NIL* as the address of the next node (which does not exist).
- If the head of the SLL is *NIL*, the list is considered empty.

Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

<u>SLLNode:</u>
  info: TElem *//the actual information*
  next: ↑ SLLNode *//address of the next node*

Singly Linked Lists - Representation

- For the representation of a SLL we need two structures: one structure for the node and one for the list itself.

<u>SLLNode:</u>
  info: TElem *//the actual information*
  next: ↑ SLLNode *//address of the next node*

<u>SLL:</u>
  head: ↑ SLLNode *//address of the first node*

- Usually, for a SLL, we only memorize the address of the head. However, there might be situations when we memorize the address of the tail as well (if the application requires it).
- If the SLL is empty (it has no elements) then the value of *head* is NULL.

SLL - Operations

- Possible operations for a singly linked list (any operation that can be done on a Dynamic Array can be done on a linked list as well):
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, after a given value)
  - delete an element (from the beginning, from the end, from a given position, with a given value)
  - get an element from a position

- These are *possible* operations; usually we need only part of them, depending on the container that we implement using a SLL.

SLL - Search

function search (sll, elem) **is:**

SLL - Search

function search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
  current ← sll.head
  while current ≠ NIL **and** [current].info ≠ elem **execute**
    current ← [current].next
  end-while
  search ← current
end-function

- Complexity:

SLL - Search

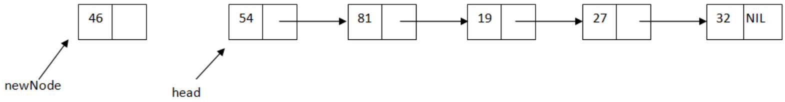
function search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
current $\leftarrow$ sll.head
while current $\neq$ NIL **and** [current].info $\neq$ elem **execute**
current $\leftarrow$ [current].next
end-while
search $\leftarrow$ current
end-function

- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.
- What happens if *sll* is empty?

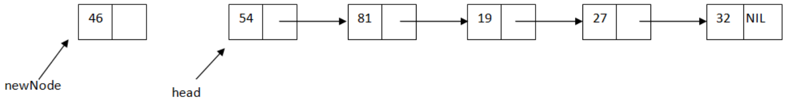
SLL - Walking through a linked list

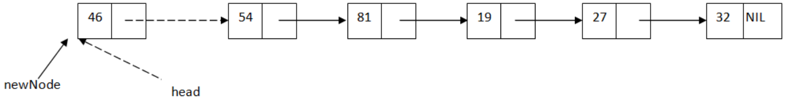
- In the *search* function we have seen how we can walk through the elements of a linked list:
  - we need an auxiliary node (called *current*), which starts at the head of the list
  - at each step, the value of the *current* node becomes the address of the successor node (through the $current \leftarrow [current].next$ instruction)
  - we stop when the current node becomes *NIL*

SLL - Insert at the beginning



SLL - Insert at the beginning





SLL - Insert at the beginning

subalgorithm insertFirst (sll, elem) **is:**

SLL - Insert at the beginning

```
subalgorithm insertFirst (sll, elem) is:
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
   newNode ← allocate() //allocate a new SLLNode
   [newNode].info ← elem
   [newNode].next ← sll.head
   sll.head ← newNode
end-subalgorithm
```

- Complexity:

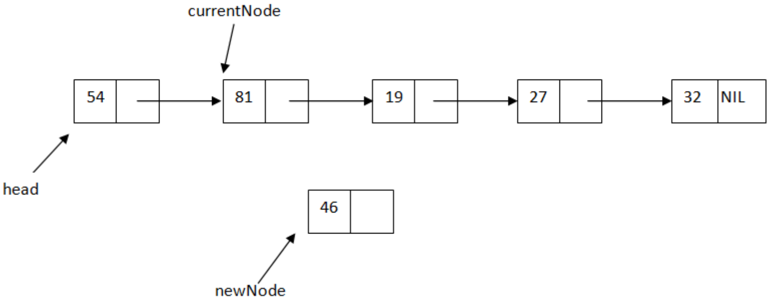
SLL - Insert at the beginning

subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: the element elem will be inserted at the beginning of sll
  newNode ← allocate() *//allocate a new SLLNode*
  [newNode].info ← elem
  [newNode].next ← sll.head
  sll.head ← newNode
end-subalgorithm

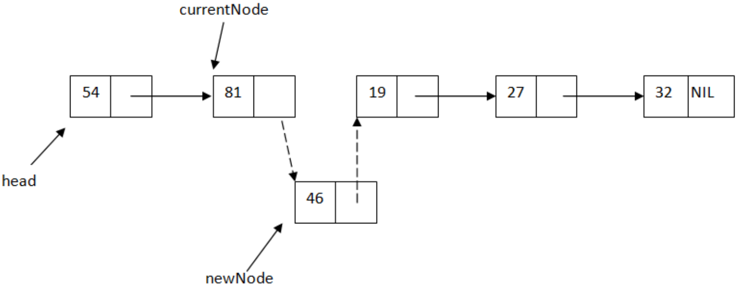
- Complexity: $\Theta(1)$
- What happens if *sll* is empty?

SLL - Insert after a node

- Suppose that we have the address of a node from the SLL and we want to insert a new element after that node.



SLL - Insert after a node



SLL - Insert after a node

subalgorithm insertAfter(sll, currentNode, elem) **is:**

SLL - Insert after a node

subalgorithm insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
  newNode ← allocate() *//allocate a new SLLNode*
  [newNode].info ← elem
  [newNode].next ←[currentNode].next
  [currentNode].next ← newNode
end-subalgorithm

- Complexity:

SLL - Insert after a node

subalgorithm insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
newNode ← allocate() *//allocate a new SLLNode*
[newNode].info ← elem
[newNode].next ←[currentNode].next
[currentNode].next ← newNode
end-subalgorithm

- Complexity: $\Theta(1)$
- What happens if *currentNode* is the first or the last node from the *sll*?