DATA STRUCTURES AND ALGORITHMS

LECTURE 5

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 4...

- Sorted Lists
- Circular Lists
- Linked Lists on Arrays

Today

1. Skip Lists
2. ADT Set
3. ADT Map
4. Iterator
5. ADT Matrix
6. Heap

Skip Lists

- Assume that we want to memorize a sequence of sorted elements. The elements can be stored in:
  - dynamic array
  - linked list
- What is the time complexity of inserting a new element into the sequence?
  - We can divide the insertion into two steps: *finding the position* and *inserting the element*.

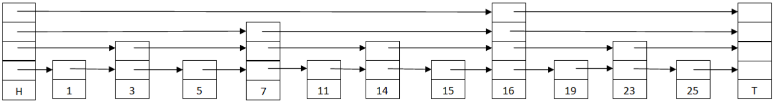
Skip List

- A skip list is a data structure that allows *fast search* in an ordered sequence.
- How can we do that?

Skip List

- A skip list is a data structure that allows *fast search* in an ordered sequence.
- How can we do that?
  - Starting from an ordered linked list, we add to every second node another pointer that skips over one element.
  - We add to every fourth node another pointer that skips over 3 elements.
  - etc.

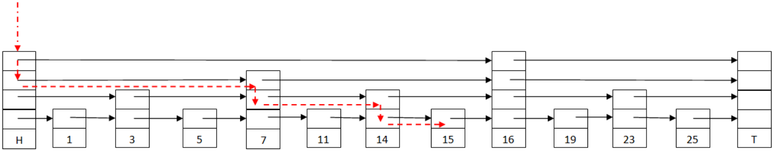
Skip List



- H and T are two special nodes, representing *head* and *tail*. They cannot be deleted, they exist even in an empty list.

Skip List - Search

- Search for element 15.



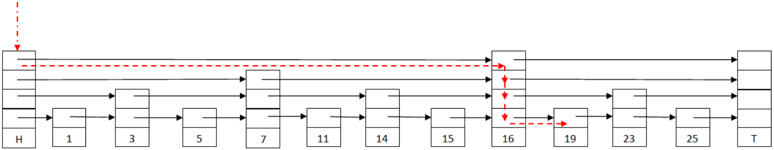
- Start from head and from highest level.
- If possible, go right.
- If cannot go right (next element is greater), go down a level.

Skip List

- Lowest level has all n elements.
- Next level has $\frac{n}{2}$ elements.
- Next level has $\frac{n}{4}$ elements.
- etc.
- $\Rightarrow$ there are approx $log_2 n$ levels.
- From each level, we check at most 2 nodes.
- Complexity of search: $O(log_2 n)$

Skip List - Insert

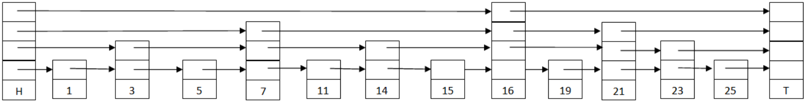
- Insert element 21.



- How *high* should the new node be?

Skip List - Insert

- *Height* of a new node is determined *randomly*, but in such a way that approximately half of the nodes will be on level 2, a quarter of them on level 3, etc.



- Assume we randomly generate the height 3 for the node with 21.

Skip List

- Skip Lists are *probabilistic* data structures, since we decide randomly the height of a newly inserted node.
- There might be a worst case, where every node has height 1 (so it is just a linked list).
- In practice, they function well.

ADT Set

- A *Set* is a container in which the elements are unique, and their order is not important (they do not have positions).
  - No operations based on positions.
  - We cannot make assumptions regarding the order in which elements are stored and will be iterated.
- Domain of the ADT Set:
  $\mathcal{S} = \{s | \text{s is a set with elements of the type TElem}\}$

Set - Interface I

- init (s)
  - **descr:** creates a new empty set.
  - **pre:** true
  - **post:** $s \in \mathcal{S}$, s is an empty set.

Set - Interface II

- add(s, e)
  - **descr:** adds a new element into the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:**$s' \in \mathcal{S}$, $s' = s \cup \{e\}$ (e is added only if it is not in s yet. If s contains the element e already, no change is made).

  - What happens if e is already in s?

Set - Interface III

- remove(s, e)
  - **descr:** removes an element from the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:** $s \ \in \mathcal{S}$, $s' = s \setminus \{e\}$ (if e is not in s, s is not changed).

Set - Interface IV

- find(s, e)
  - **descr:** verifies if an element is in the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:**

$$find \leftarrow \begin{cases} True, & \text{if } e \in s \\ False, & \text{otherwise} \end{cases}$$

Set - Interface V

- size(s)
  - **descr:** returns the number of elements from a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** size $\leftarrow$ the number of elements from s

Set - Interface VI

- iterator(s, it)
  - **descr:** returns an iterator for a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over the set s

Set - Interface VII

- destroy (s)
    - **descr:** destroys a set
    - **pre:** $s \in S$
    - **post:** the set s was destroyed.

Set - Interface VIII

- Other possible operations (characteristic for sets from mathematics):
  - reunion of two sets
  - intersection of two sets
  - difference of two sets (elements that are present in the first set, but not in the second one)

Sorted Set

- We can have a Set where the elements are ordered based on a *relation* $\rightarrow$ *SortedSet*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted set, the iterator has to iterate through the elements in the order given by the *relation*.

Set

- If we want to implement the ADT Set (or ADT SortedSet), we can use the following data structures as representation:
    - (dynamic) array
    - linked list
    - hash tables - to be discussed later
    - (balanced) binary trees - for sorted sets - to be discussed later
    - skip lists - for sorted sets

ADT Map

- A *Map* is a container where the elements are <*key, value*> pairs.
- Each *key* has one single associated *value*, and we can access the values only by using the key $\rightarrow$ no positions in a *Map*.
- Keys have to be unique in a *Map*, and each *key* has one single associated value (if a key can have multiple values we have a *MultiMap*).
- When we implement a *Map*, we should use a data structure that makes finding the *keys* easy.

Map

- Examples of using a map:
  - Bank account number (as key) and every information associated with the bank account (as value)
  - Student id (as key) and every information about the student (as value)
  - etc.
- Domain of the ADT Map:

$$\mathcal{M} = \{m | \text{m is a map with elements } e = (k, v), \text{ where } k \in TKey \text{ and } v \in TValue\}$$

Map - Interface I

- init(m)
  - **descr:** creates a new empty map
  - **pre:** true
  - **post:** $m \in \mathcal{M}$, m is an empty map.

Map - Interface II

- destroy(m)
  - **descr:** destroys a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** m was destroyed

Map - Interface III

- add(m, k, v)
  - **descr:** add a new key-value pair to the map (the operation can be called *put* as well)
  - **pre:** $m \in \mathcal{M}, k \in TKey, v \in TValue$
  - **post:** $m' \in \mathcal{M}, m' = m \cup < k, v >$

  - What happens if there is already a pair with k as key?

Map - Interface IV

- remove(m, k, v)
  - **descr:** removes a pair with a given key from the map
  - **pre:** $m \in \mathcal{M}, k \in TKey$
  - **post:** $v \in TValue$, where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \text{ and } m' \in \mathcal{M}, \\ & m' = m \setminus < k, v' > \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

Map - Interface V

- search(m, k, v)
  - **descr:** searches for the value associated with a given key in the map
  - **pre:** $m \in \mathcal{M}, k \in TKey$
  - **post:** $v \in TValue$, where

$$v \leftarrow \begin{cases} v', & \text{if } \exists < k, v' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

Map - Interface VI

- iterator(m, it)
  - **descr:** returns an iterator for a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over m.

Map - Interface VII

- size(m)
  - **descr:** returns the number of pairs from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** size $\leftarrow$ the number of pairs from m

Map - Interface VIII

- keys(m, s)
  - **descr:** returns the set of keys from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from m

Map - Interface IX

- values(m, b)
    - **descr:** returns a bag with all the values from the map
    - **pre:** $m \in \mathcal{M}$
    - **post:**$b \in \mathcal{B}$, b is the bag of all values from m

Map - Interface X

- pairs(m, s)
  - **descr:** returns the set of pairs from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $s \in \mathcal{S}$, s is the set of all pairs from m

Sorted Map

- We can have a Map where we can define an order (a relation) on the set of possible keys: instead of *TKey* we will have *TComp*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSets.

Map

- If we want to implement the ADT Map (or ADT SortedMap), we can use the following data structures as representation:
  - (dynamic) array
  - linked list
  - hash tables - to be discussed later
  - (balanced) binary trees - for sorted maps - to be discussed later
  - skip lists - for sorted maps

Iterator - why do we need it? I

- Most containers have iterators and for every data structure we will discuss how we can implement an iterator for a container defined on that data structure.
- Why are iterators so important?

Iterator - why do we need it? II

- They offer a uniform way of iterating through the elements of any container

Iterator - why do we need it? III

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
   iterator(c, it)
   while valid(it) execute
      //get the current element from the iterator
      getCurrent(it, elem)
      print elem
      //go to the next element
      next(it)
   end-while
end-subalgorithm
```

Iterator - why do we need it? IV

- For most containers the iterator is the only thing we have to *see* the content of the container.
    - List (will be discussed later) is the only container that has positions, for other containers we can use only the iterator.

Iterator - why do we need it? V

- Giving up positions, we can gain performance.
  - Containers that do not have positions can be represented on data structures where some operations have good complexities, but where the notion of a position does not naturally exist and where enforcing positions is really complicated (ex. hash tables).

Iterator - why do we need it? VI

- Even if we have positions, using an iterator might be faster.
  - Going through the elements of a linked list with an iterator is faster than going through every position one-by-one.

ADT Matrix

- A *Matrix* is a container that represents a two-dimensional array.
- Each element has a unique position, determined by two indexes: its line and column.
- The operations for a Matrix are different from the operations that exist for most other containers, because in a Matrix we cannot add elements, and we cannot delete an element from a Matrix, we can only change the value of an element.

Matrix - Operations

- The minimum set of operations that should exist for the ADT Matrix is:
  - init(matrix, nrL, nrC) - create a new matrix with *nrL* lines and *nrC* columns
  - nrLine(matrix) - return the number of lines from the matrix
  - nrColumns(matrix) - return the number of columns from the matrix
  - element(matrix, i, j) - return the element from the line *i* and column *j*
  - modify(matrix, i, j, val) - change the values of the element from line *i* and column *j* into *val*

Matrix - Operations

- Other possible operations:
  - get the position of a given element
  - create an iterator that goes through the elements by columns
  - create an iterator the goes through the elements by lines
  - etc.

Matrix - representation

- Usually a sequential representation is used for a Matrix (we memorize all the lines one after the other in a consecutive memory block).
- If the Matrix contains many values of 0 (or 0_{TElem}), we have a *sparse matrix*, where it is more (space) efficient to memorize only the elements that are different from 0.

Sparse Matrix example

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 5 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 \\ 1 & 0 & 0 & 7 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 5 \\ 0 & 0 & 9 & 1 & 0 & 0 \end{bmatrix}$$

- Out of the 36 elements, only 10 are different from 0.

Sparse Matrix - representation

- We can memorize (line, column, value) triples, where value is different from 0 (or 0_{TElem}). For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column).
- Triples can be stored in:
  - (dynamic) arrays
  - linked lists
  - (balanced) binary trees

Sparse Matrix - representation example

- For the previous example we would keep the following triples: $< 1, 3, 3 >$, $< 1, 5, 5 >$, $< 2, 1, 2 >$, $< 3, 6, 4 >$, $< 4, 1, 1 >$, $< 4, 4, 7 >$, $< 5, 2, 6 >$, $< 5, 6, 5 >$, $< 6, 3, 9 >$, $< 6, 4, 1 >$.
- We need to retain the dimensions of the matrix as well (we might have last line(s) or column(s) with only 0 values).

Sparse Matrix - representation

- Linked representation, using circular lists.
- Each node contains the line, the column, and the value (different from 0) and each node has two pointers: to the next element on the same line and to the next element on the same column. Last elements keep a pointer to the first ones (circular lists).
- We will have special nodes for each line and each column to show the beginning of the corresponding list.

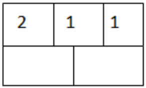
Sparse Matrix - representation example

- For the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 7 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & 9 & 1 \end{bmatrix}$$

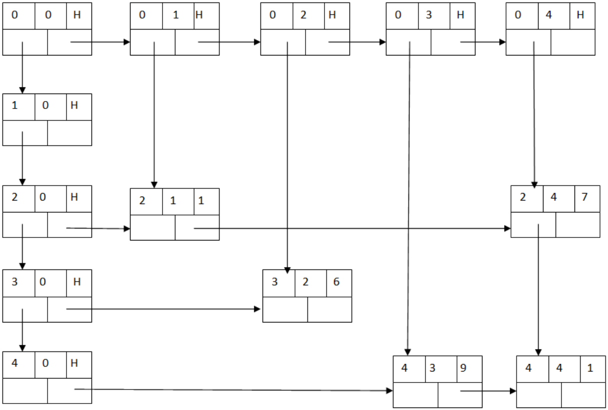
Sparse Matrix - representation example

- The linked lists will be made of nodes. Each node contains the line, column and value and two pointers: one to the next element on the same line, and one to the next element from the same column.



- This is how a node will be represented on the following figures. It represents the element from line 2, column 1 with value 1.

Sparse Matrix - representation example



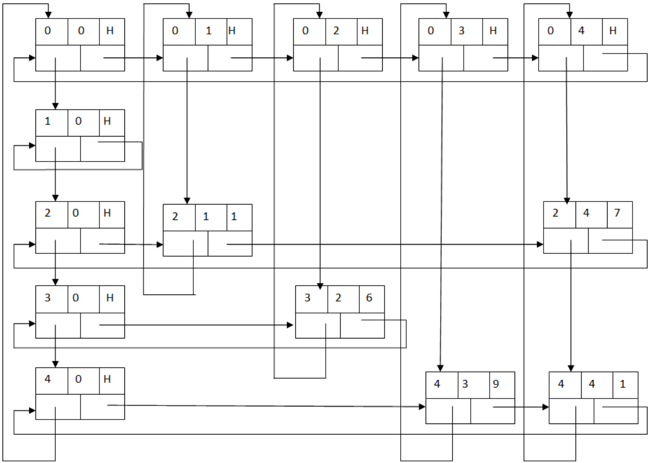
- Nodes with line or column 0 and with value H, are *header* nodes, they do not represent actual elements, just the first node of the corresponding column or row.

- Obviously, the nodes can be anywhere in the memory (but it is easier to understand the representation if we draw them like this).

- And since we have circular lists, on each row and column the last node has a pointer to the corresponding header node.

- It is enough to retain the address of header node 0,0.

Sparse Matrix - representation example



Sparse Matrix - operations

- Operations of a sparse matrix are exactly the same as the operations for a *regular* matrix. The most difficult operation is *modify*, because here we have 4 different cases, based on the current value at line i and column j (we will call it *old_value*) and the value we want to put there (*new_value*).

Sparse Matrix - operations

- Operations of a sparse matrix are exactly the same as the operations for a *regular* matrix. The most difficult operation is *modify*, because here we have 4 different cases, based on the current value at line i and column j (we will call it *old_value*) and the value we want to put there (*new_value*).
  - $old\_value = 0$ and $new\_value = 0 \Rightarrow$ do nothing
  - $old\_value = 0$ and $new\_value \neq 0 \Rightarrow$ add a new triple/node with *new_value*
  - $old\_value \neq 0$ and $new\_value = 0 \Rightarrow$ delete the triple/node with *old_value*
  - $old\_value \neq 0$ and $new\_value \neq 0 \Rightarrow$ modify the value from the triple/node to *new_value*

Heap

- A binary heap is a data structure that can be used as an efficient representation for Priority Queues (will be discussed later).
- A binary heap is a kind of hybrid between a dynamic array and a binary tree.
- The elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree.

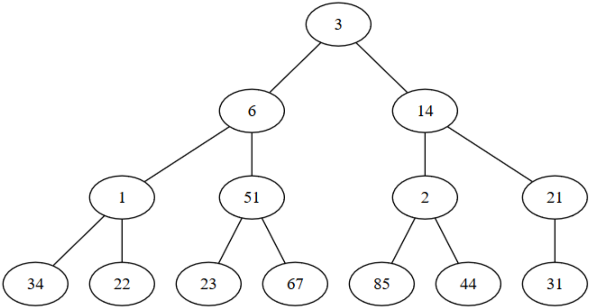
Heap

- Assume that we have the following array (upper row contains positions, lower row contains elements):

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31

Heap

- We can visualize this array as a binary tree, in which each node has exactly 2 children, except for the last two rows, but there the children of the nodes are completed from left to right.



Heap

- If the elements of the array are: $a_1, a_2, a_3, \ldots, a_n$, we know that:
  - a_1 is the root of the heap
  - for an element from position i, its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than n)
  - for an element from positions i (i $>$ 1), the parent of the element is on position $[i/2]$ (integer part of i/2)

Heap

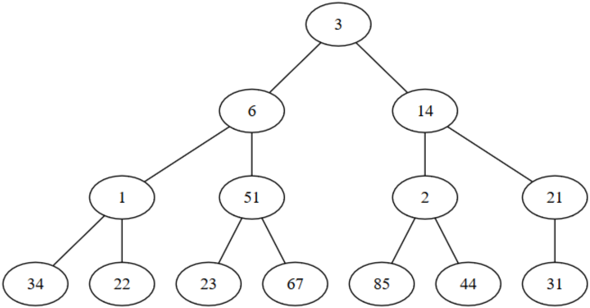
- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.

  - *Heap structure:* in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.

  - *Heap property:* $a_i \geq a_{2*i}$ (if $2*i \leq n$) and $a_i \geq a_{2*i+1}$ (if $2*i+1 \leq n$)
  - The $\geq$ relation between a node and both its descendants can be generalized (other relations can be used as well).

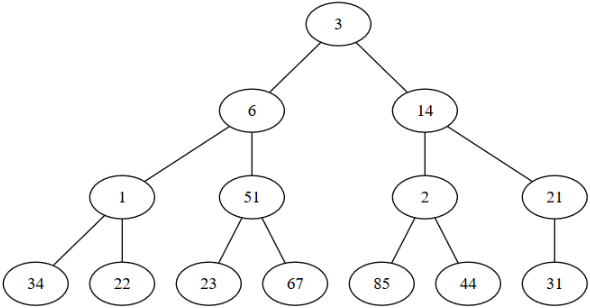
Heap

- Is this a heap?



Heap

- Is this a heap?



- No. It has the heap structure, but it does not have the heap property.

Heap - Notes

- If we use the $\geq$ relation, we will have a *MAX-HEAP*.
- If we use the $\leq$ relation, we will have a *MIN-HEAP*.
- The height of a heap with n elements is $log_2 n$, so the operations performed on the heap have $O(log_2 n)$ complexity.

Heap - operations

- A heap can be used as representation for a Priority Queue and it has two specific operations:
    - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
    - remove (we always remove the root of the heap - no other element can be removed).