DATA STRUCTURES AND ALGORITHMS

LECTURE 11

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 10...

- Hash tables
  - Separate chaining
  - Coalesced chaining
  - Open Addressing

Today

1. Trees

Open addressing - review

- For open addressing we place the elements directly in the hash table.
- We change the hash function to take as parameter a *probe number* as well, which is a number between 0 and $m-1$. We always start with position $h(k,0)$ and if that is occupied, we increment the probe number ($h(k,1)$, $h(k,2)$, etc.) until we either find an empty location or the probe number becomes $m-1$ and we consider that the table is full.
- We discussed three possibilities to define $h(k,i)$:
  - Linear probing
  - Quadratic probing
  - Double hashing

Open addressing - operations

- In the following we will discuss the implementation of the basic dictionary operations for collision resolution with open addressing.
- We will use the notation $h(k, i)$ for a hash function, without mentioning whether we have linear probing, quadratic probing or double hashing (code is the same for each of them, implementation of h is different only).

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

HashTable:
T: TKey[]
m: Integer
h: TFunction

- For simplicity we will consider that we only have keys.

Open addressing - insert

- What should the *insert* operation do?

Open addressing - insert

- What should the *insert* operation do?

```
subalgorithm insert (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash
    else
        ht.T[pos] ← e
   end-if
end-subalgorithm
```

Open addressing - search

- What should the *search* operation do?

Open addressing - search

- What should the *search* operation do?

```
function search (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: search returns true if e is in ht and false otherwise
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 and ht.T.[pos] ≠ e execute
    //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i < ht.m and ht.T[pos] = e execute
        search ← True
    else
        search ← False
    end-if
end-function
```

Open addressing - remove

- How can we remove an element from the hash table?

Open addressing - remove

- How can we remove an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements

Open addressing - remove

- How can we remove an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

Open addressing - Performance

- In a hash table with open addressing with load factor $\alpha = n/m$ ($\alpha < 1$), the *average* number of probes is at most
  - for *insert* and *unsuccessful search*

$$\frac{1}{1-\alpha}$$

  - for *successful search*

$$\frac{1}{\alpha} * ln\frac{1}{1-\alpha}$$

- If α is constant, the complexity is $\Theta(1)$
- Worst case complexity is $\Theta(n)$

Perfect hashing

Perfect hashing

- Assume that we know all the keys in advance and we use *separate chaining* for collision resolution $\Rightarrow$ the more lists we make, the shorter the lists will be (reduced number of collisions) $\Rightarrow$ if we could make a large number of list, each would have one element only (no collision).
- How large should we make the hash table to make sure that there are no collisions?
- If $M = N^2$, it can be shown that the table is collision free with probability at least $1/2$.
- Start building the hash table. If you detect a collision, just choose a new hash function and start over (expected number of trials is at most 2).

Perfect hashing

- Having a table of size N^2 is impractical.
- Solution instead:
  - Use a hash table of size N (*primary* hash table).
  - Instead of using linked list for collision resolution (as in separate chaining) each element of the hash table is another hash table (*secondary hash table*)
  - Make the secondary hash table of size ${n_j}^2$, where n_j is the number of elements from this hash table.
  - Each secondary hash table will be constructed with a different hash function, and will be reconstructed until it is collision free.
- This is called **perfect hashing**.
- It can be shown that the total space needed for the secondary hash tables is at most $2N$.

Perfect hashing

- Perfect hashing requires multiple hash functions, this is why we use *universal hashing*.
- Let p be a prime number, larger than the largest possible key.
- The universal hash function family $\mathcal{H}$ can be defined as:

$$\mathcal{H} = \{H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m),$$

$$\text{where } 1 \leq a \leq p-1, 0 \leq b \leq p-1$$

- a and b are chosen randomly when the hash function is initialized.

Perfect hashing - example

- Insert into a hash table with perfect hashing the letters from "PERFECT HASHING EXAMPLE". Since we want no collisions at all, we are going to consider only the unique letters: "PERFCTHASINGXML"
- Since we are inserting $N = 15$ elements, we will take $m = 15$.
- For each letter, the *hashCode* is the index of the letter in the alphabet.

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12

Perfect hashing - example

- p has to be a prime number larger than the maximum key $\Rightarrow$ 29
- The hash function will be:

$$H_{a,b}(x) = ((a * x + b) \ \% \ p) \ \% \ m$$

- where a will be 3 and b will be 2 (chosen randomly).

Letter	P	E	R	F	C	T	H	A	S	I	N	G	X	M	L
HashCode	16	5	18	6	3	20	8	1	19	9	14	7	24	13	12
H(HashCode)	6	2	12	5	11	4	11	5	1	0	0	8	1	12	9

Perfect hashing - example

- Occupied positions/Collisions:
  - position 0 - I, N
  - position 1 - S, X
  - position 2 - E
  - position 4 - T
  - position 5 - F, A
  - position 6 - P
  - position 8 - G
  - position 9 - L
  - position 11 - C, H
  - position 12 - R, M

Perfect hashing - example

- For the positions where we have no collision (only one element hashed to it) we will have a secondary hash table with only one element, and hash function $h(x) = 0$
- For the positions where we have two elements, we will have a secondary hash table with 4 positions and different hash functions, taken from the same universe, with different random values for a and b.
- For example for position 0, we can define $a = 4$ and $b = 11$ and we will have:
  $h(I) = h(9) = 2$
  $h(N) = h(14) = 1$

Perfect hashing - example

- Assume that for the secondary hash table from position 1 we will choose $a = 5$ and $b = 2$.
- Positions for the elements will be:
  $h(S) = h(19) = ((5 * 19 + 2)\%29)\%4 = 2$
  $h(X) = h(24) = ((5 * 24 + 2)\%29)\%4 = 2$
- In perfect hashing we should not have collisions, so we will simply chose another hash function: another random values for a and b. Choosing for example $a = 2$ and $b = 13$, we will have h(S) = 2 and h(X) = 3.

Perfect hashing

- When perfect hashing is used and we search for an element we will have to check at most 2 positions (position in the primary and in the secondary table).
- This means that worst case performance of the table is $\Theta(1)$.
- But in order to use perfect hashing, we need to have static keys: once the table is built, no new elements can be added.

Cuckoo hashing

Cuckoo hashing

- In cuckoo hashing we have two hash tables of the same size, each of them more than half empty and each hash table has its hash function (so we have two different hash functions).
- For each element to be added we can compute two positions: one from the first hash table and one from the second. In case of cuckoo hashing, it is guaranteed that an element will be on one of these positions.
- Search is simple, because we only have to look at these two positions.
- Delete is simple, because we only have to look at these two positions and set to empty the one where we find the element.

Cuckoo hashing

- When we want to insert a new element we will compute its position in the first hash table. If the position is empty, we will place the element there.
- If the position in the first hash table is not empty, we will kick out the element that is currently there, and place the new element into the first hash table.
- The element that was kicked out, will be placed at its position in the second hash table. If that position is occupied, we will kick out the element from there and place it into its position in the first hash table.
- We repeat the above process until we will get an empty position for an element.
- If we get back to the same location with the same key we have a cycle and we cannot add this element $\Rightarrow$ resize, rehash

Cuckoo hashing - example

- Assume that we have two hash tables, with $m = 11$ positions and the following hash functions:
  - h1(k) = k % 11
  - h2(k) = (k div 11) % 11

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example I

- Insert key 20
  - h1(20) = 9 - empty position, element added in the first table
- Insert key 50
  - h1(50) = 6 - empty position, element added in the first table

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo hashing - example II

- Insert key 53
  - h1(53) = 9 - occupied
  - 53 goes in the first hash table, and it sends 20 in the second to position h2(20) = 1

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									

Cuckoo hashing - example III

- Insert key 75
  - h1(75) = 9 - occupied
  - 75 goes in the first hash table, and it sends 53 in the second to position h2(53) = 4

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

Cuckoo hashing - example IV

- Insert key 100
  - h1(100) = 1 - empty position
- Insert key 67
  - h1(67) = 1 - occupied
  - 67 goes in the first hash table, and it sends 100 in the second to position h2(100) = 9

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo hashing - example V

- Insert key 105
  - h1(105) = 6 - occupied
  - 105 goes in the first hash table, and it sends 50 in the second to position h2(50) = 4
  - 50 goes in the second hash table, and it sends 53 to the first one, to position h1(53) = 9
  - 53 goes in the first hash table, and it sends 75 to the second one, to position h2(75) = 6

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

Cuckoo hashing - example VI

- Insert key 3
  - h1(3) = 3 - empty position
- Insert key 36
  - h1(36) = 3 - occupied
  - 36 goes in the first hash table, and it sends 3 in the second to position h2(3) = 0

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

Cuckoo hashing - example VII

- Insert key 39
  - h1(39) = 6 - occupied
  - 39 goes in the first hash table and it sends 105 in the second to position h2(105) = 9
  - 105 goes to the second hash table and it sends 100 in the first to position h1(100) = 1
  - 100 goes in the first hash table and it sends 67 in the second to position h2(67) = 6
  - 67 goes in the second hash table and it sends 75 in the first to position h1(75) = 9
  - 75 goes in the first hash table and it sends 53 in the second to position h2(53) = 4
  - 53 goes in the second hash table and it sends 50 in the first to position h1(50) = 6
  - 50 goes in the first hash table and it sends 39 in the second to position h2(39) = 3

Cuckoo hashing - example VIII

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

Cuckoo hashing

- It can happen that we cannot insert a key because we get in a cycle. In these situation we have to increase the size of the tables and rehash the elements.
- While in some situation insert moves a lot of elements, it can be shown that if the load factor of the tables is below 0.5, the probability of a cycles is low and it is very unlikely that more than $O(log_2 n)$ elements will be moved.

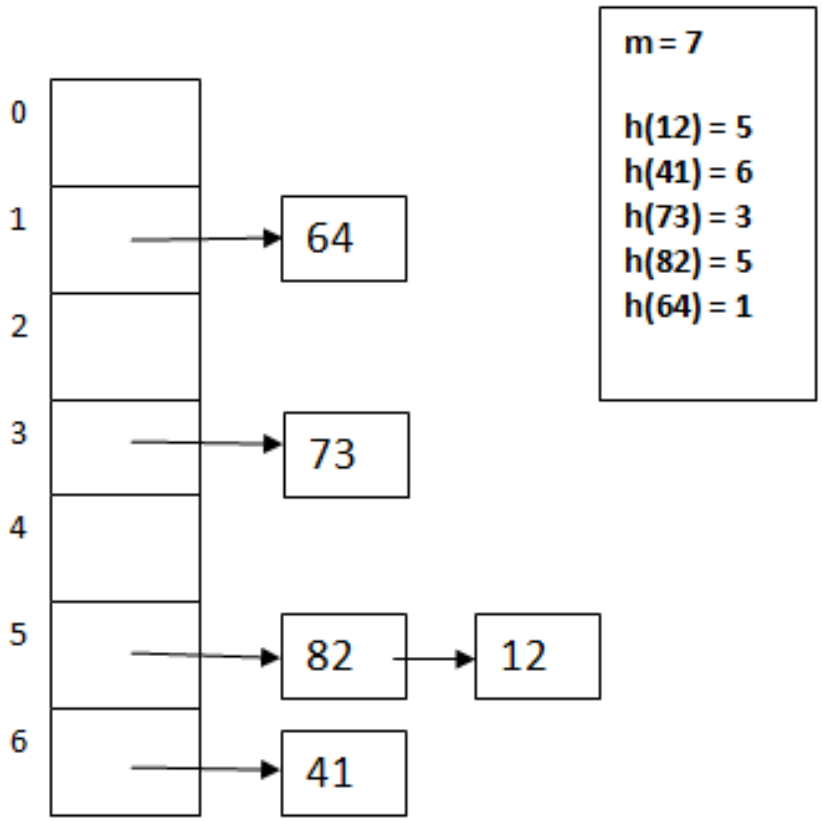
Cuckoo hashing

- If we use two tables and each position from a table holds one element at most, the tables have to have load factor below 0.5 to work well.
- If we use three tables, the tables can have load factor of 0.91 and for 4 tables we have 0.97

Linked Hash Table

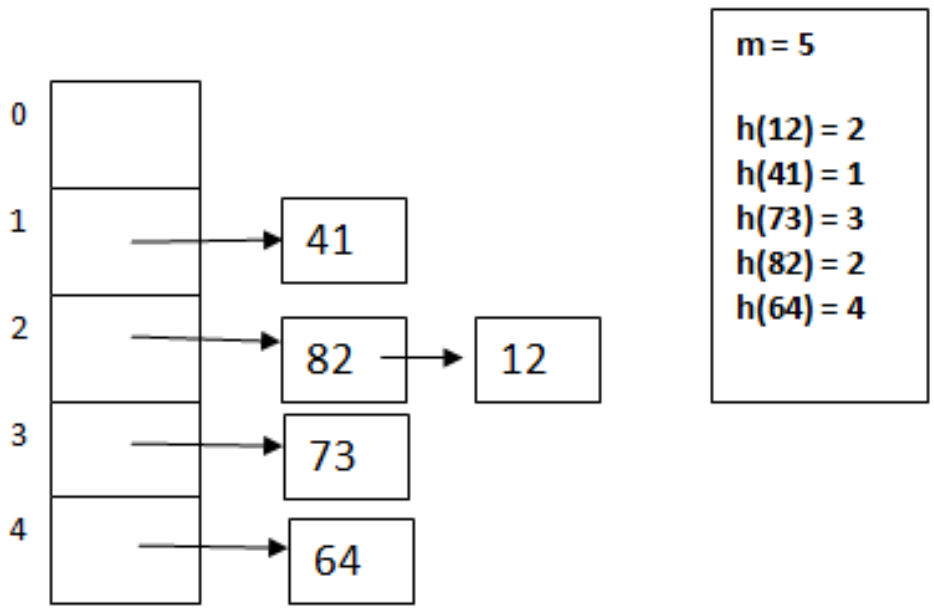
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

Linked Hash Table



- Iteration order: 41, 82, 12, 73, 64

Linked Hash Table

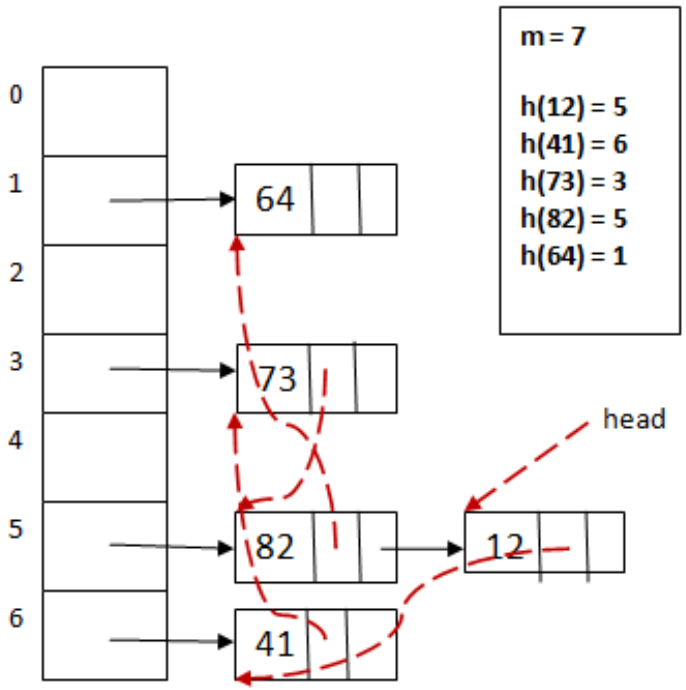
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

Node:
info: TKey
nextH: ↑ Node *//pointer to next node from the collision*
nextL: ↑ Node *//pointer to next node from the insertion-order list*
prevL: ↑ Node *//pointer to prev node from the insertion-order list*

LinkedHT:
m:Integer
T:(↑ Node)[]
h:TFunction
head: ↑ Node
tail: ↑ Node

Linked Hash Table - Insert

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...
```

Linked Hash Table - Insert

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

Linked Hash Table - Remove

- How can we implement the *remove* operation?

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current ≠ NIL then
            nodeToBeRemoved ← current
            [prevNode].nextH ← [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head ← NIL
                lht.tail ← NIL
            else
                lht.head ← [lht.head].nextL
                [lht.head].prev ← NIL
            end-if
//continued on the next slide...
```

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```

Trees

- Trees are one of the most commonly used data structures because they offer an efficient way of storing data and working with the data.
- In graph theory a *tree* is a connected, acyclic graph (usually undirected).
- When talking about trees as a data structure, we actually mean *rooted trees*, trees in which one node is designated to be the *root* of the tree.

Tree - Definition

- A tree is a finite set $\mathcal{T}$ of 0 or more elements, called *nodes*, with the following properties:
  - If $\mathcal{T}$ is empty, then the tree is empty
  - If $\mathcal{T}$ is not empty then:
    - There is a special node, R, called the *root* of the tree
    - The rest of the nodes are divided into k ($k \geq 0$) disjunct *trees*, T_1, T_2, ..., T_k, the root node R being linked by an edge to the root of each of these trees. The trees T_1, T_2, ..., T_k are called the *subtrees* (*children*) of R, and R is called the *parent* of the subtrees.

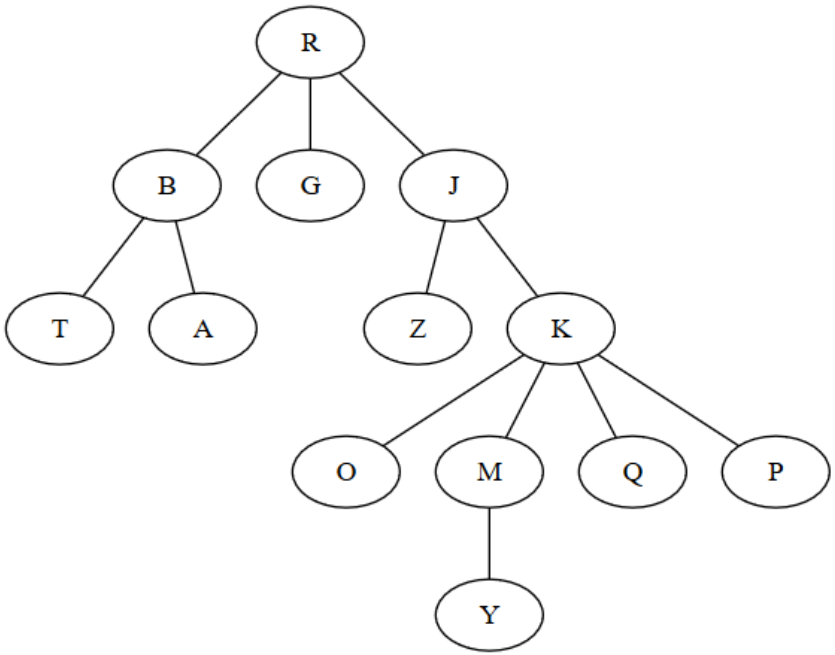
Tree - Terminology I

- An *ordered tree* is a tree in which the order of the children is well defined and relevant (instead of having a set of children, each node has a list of children).
- The *degree* of a node is defined as the number of children of the node.
- The nodes with the degree 0 (nodes without children) are called *leaf nodes*.
- The nodes that are not leaf nodes are called *internal nodes*.

Tree - Terminology II

- The *depth* or *level* of a node is the length of the path (measured as the number of edges traversed) from the root to the node. This path is unique. The root of the tree is at level 0 (and has depth 0).
- The *height* of a node is the length of the longest path from the node to a leaf node.
- The *height of the tree* is defined as the height of the root node, i.e., the length of the longest path from the root to a leaf.

Tree - Terminology Example



- Root of the tree: R
- Children of R: B, G, J
- Parent of M: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node K: 2 (path R-J-K)
- Height of node K: 2 (path K-M-Y)
- Height of the tree (height of node R): 4
- Nodes on level 2: T, A, Z, K

k-ary trees

- How can we represent a tree in which every node has at most k children?
- One option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - k fields, one for each child

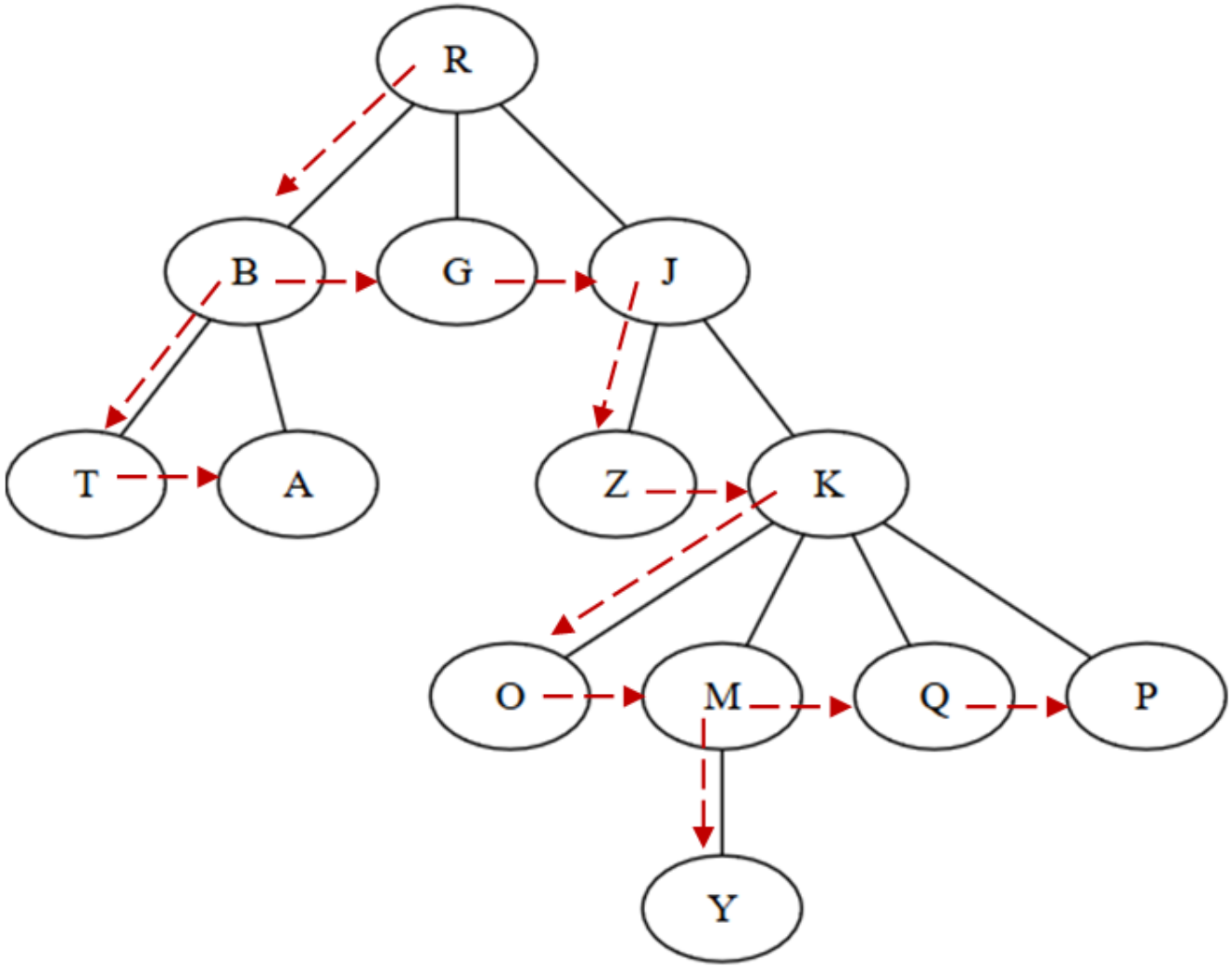
k-ary trees

- Another option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - an array of dimension k, in which each element is the address of a child
  - number of children (number of occupied positions from the above array)
- Disadvantage of these approaches is that we occupy space for k children even if most nodes have less children.

k-ary trees

- A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - address of the leftmost child of the node
  - address of the right sibling of the node (next node on the same level from the same parent).

Left-child right sibling representation example



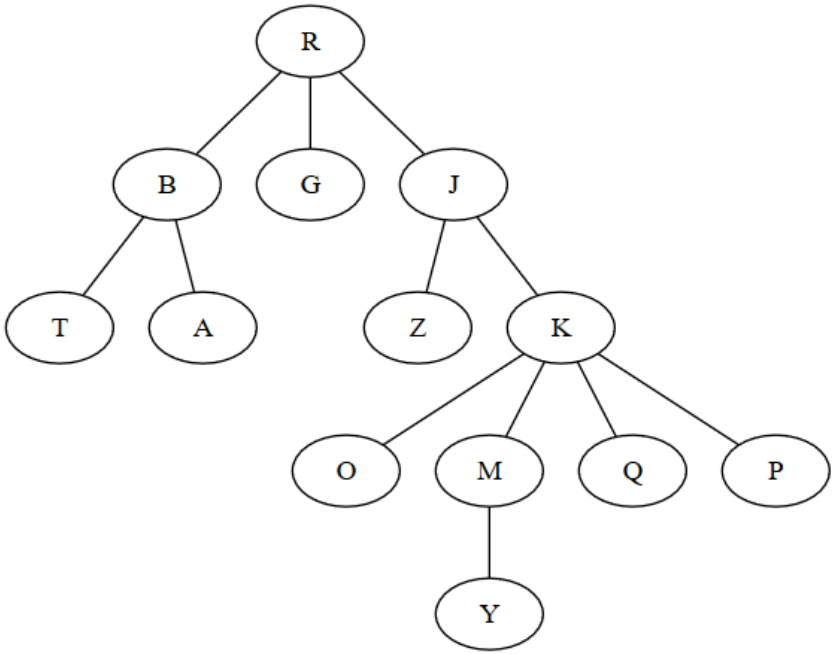
Tree traversals

- A node of a tree is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on the node (printing it, checking the value from the node, etc.).
- *Traversing* a tree means visiting all of its nodes.
- For a k-ary tree there are 2 possible traversals:
  - Depth-first traversal
  - Level order (breadth first) traversal

Depth first traversal

- Traversal starts from root
- From root we visit one of the children, than one child of that child, and so on. We go down (in depth) as much as possible, and continue with other children of a node only after all descendants of the "first" child were visited.
- For depth first traversal we use a stack to remember the nodes that have to be visited.

Depth first traversal example

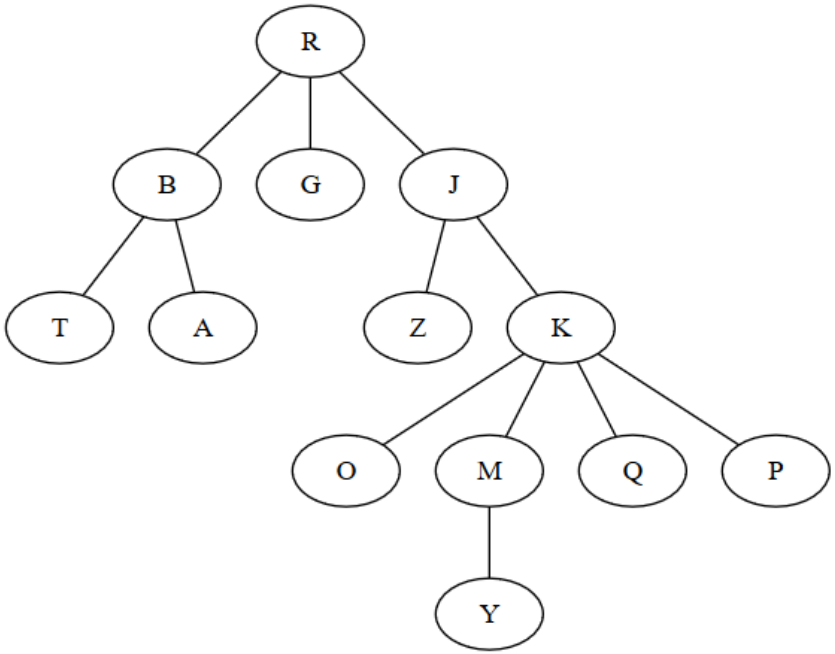


- Stack s with the root: R
- Visit R (pop from stack) and push its children: $s = [\text{B G J}]$
- Visit B and push its children: $s = [\text{T A G J}]$
- Visit T and push nothing: $s = [\text{A G J}]$
- Visit A and push nothing: $s = [\text{G J}]$
- Visit G and push nothing: $s = [\text{J}]$
- Visit J and push its children: $s = [\text{Z K}]$
- etc...

Level order traversal

- Traversal starts from root
- We visit all children of the root (one by one) and once all of them were visited we go to their children and so on. We go down one level, only when all nodes from a level were visited.
- For level order traversal we use a queue to remember the nodes that have to be visited.

Level order traversal example



- Queue q with the root: R
- Visit R (pop from queue) and push its children: q = [B G J]
- Visit B and push its children: q = [G J T A]
- Visit G and push nothing: q = [J T A]
- Visit J and push its children: q = [T A Z K]
- Visit T and push nothing: q = [A Z K]
- Visit A and push nothing: q = [Z K]
- etc...