DATA STRUCTURES

LECTURE 11

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 10...

- Hash tables
  - Hash function
  - Separate chaining

Today

- Coalesced chaining
- Open addressing
- Trees

Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.
- When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the *next* link, so that the element can be found in a search.
- Since elements are in the table, α can be at most 1.

Coalesced chaining - example

- Consider a hash table of size $m = 16$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Example

- Initially the hash table is empty. All next values are -1 and the first empty position is position 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

- 76 will be added to position 12. But 12 should also be added there. Since that position is already occupied, we add 12 to position firstEmpty and set the next of 76 to point to position 0. Then we reset firstEmpty to the next empty position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12												76			
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

Example

- And we continue in the same manner. We have no collisions up to 81, but we need to reset firstEmpty when we *accidentally* occupy it.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18				22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 3

- When adding 91, we put it to position firstEmpty and set the next link of position 11 to position 3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91			22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	3	0	-1	-1	-1

firstEmpty = 4

Example

- The final table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91	27	13	22	55	16	39		43	76	109		
8	-1	-1	4	-1	-1	-1	9	-1	-1	-1	3	0	5	-1	-1

firstEmpty = 10

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

HashTable:
- T: TKey[]
- next: Integer[]
- m: Integer
- firstEmpty: Integer
- h: TFunction

- For simplicity, in the following, we will consider only the keys.

Coalesced chaining - insert

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //-1 means empty position
        ht.T[pos] ← k
        ht.next[pos] ← -1
    else
        if ht.firstEmpty = ht.m then
            @resize and rehash
        end-if
        current ← pos
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
//continued on the next slide...
```

Coalesced chaining - insert

```
        ht.T[ht.firstEmpty] ← k
        ht.next[ht.firstEmpty] ← - 1
        ht.next[current] ← ht.firstEmpty
        changeFirstEmpty(ht)
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

Coalesced chaining - ChangeFirstEmpty

subalgorithm changeFirstEmpty(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
while ht.firstEmpty $<$ ht.m **and** ht.T[ht.firstEmpty] $\neq$ -1
execute
ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
end-while
end-subalgorithm

- Complexity: $O(m)$
- *Think about it:* Should we keep the free spaces linked in a list as in case of a linked lists on array?

Coalesced chaining

- *Remove* and *search* operations for coalesced chaining will be discussed in Seminar 6.
- How can we define an iterator for a hash table with coalesced chaining? What should the following operations do?
  - init
  - getCurrent
  - next
  - valid
- How can we implement a sorted container on a hash table with coalesced chaining? How can we implement its iterator?

Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.
- When we want to insert a new element, we will successively generate positions for the element, check (*probe*) the generated position, and place the element in the first available one.

Open addressing

- In order to generate multiple positions, we will extend the hash function and add to it another parameter, i, which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

- For an element k, we will successively examine the positions $< h(k, 0), h(k, 1), h(k, 2), ..., h(k, m-1) >$ - called the *probe sequence*
- The *probe sequence* should be a permutation of a hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.
- We would also like to have a hash function which can generate all the m! permutations possible (spoiler alert: we cannot)

Open addressing - Linear probing

- One version of defining the hash function is to use linear probing:

$$h(k,i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)

- the *probe sequence* for linear probing is: $< h'(k), h'(k)+1, h'(k)+2, ..., m-1, 0, 1, ..., h'(k)-1 >$

Open addressing - Linear probing - example

- Consider a hash table of size $m = 16$ that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key for i = 0:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

- Disadvantages of linear probing:
  - There are only m distinct probe sequences (once you have the starting position everything is fixed)
  - *Primary clustering* - long runs of occupied slots
- Advantages of linear probing:
  - Probe sequence is always a permutation
  - Can benefit from caching

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

- Worst case arrangement: all n elements are one after the other (assume in the second half of the array)
- What is the average number of probes (positions verified) that need to be checked to insert a new element?

Open addressing - Quadratic probing

- In case of quadratic probing the hash function becomes:

$$h(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$) and c_1 and c_2 are constants initialized when the hash function is initialized. c_2 should not be 0.

- Considering a simplified version of $h(k,i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be: $< k, k+1, k+4, k+9, k+16, ... >$

Open addressing - Quadratic probing

- One important issue with quadratic probing is how we can choose the values of m, c_1 and c_2 so that the probe sequence is a permutation.
- If m is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
  - For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
  - For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

Open addressing - Quadratic probing

- If m is a power of 2 and $c_1 = c_2 = 0.5$, the probe sequence will always be a permutation. For example for $m = 8$ and $k = 3$:
  - $h(3,0) = (3 \,\%\, 8 + 0.5 * 0 + 0.5 * 0^2) \,\%\, 8 = 3$
  - $h(3,1) = (3 \,\%\, 8 + 0.5 * 1 + 0.5 * 1^2) \,\%\, 8 = 4$
  - $h(3,2) = (3 \,\%\, 8 + 0.5 * 2 + 0.5 * 2^2) \,\%\, 8 = 6$
  - $h(3,3) = (3 \,\%\, 8 + 0.5 * 3 + 0.5 * 3^2) \,\%\, 8 = 1$
  - $h(3,4) = (3 \,\%\, 8 + 0.5 * 4 + 0.5 * 4^2) \,\%\, 8 = 5$
  - $h(3,5) = (3 \,\%\, 8 + 0.5 * 5 + 0.5 * 5^2) \,\%\, 8 = 2$
  - $h(3,6) = (3 \,\%\, 8 + 0.5 * 6 + 0.5 * 6^2) \,\%\, 8 = 0$
  - $h(3,7) = (3 \,\%\, 8 + 0.5 * 7 + 0.5 * 7^2) \,\%\, 8 = 7$

Open addressing - Quadratic probing

- If m is a prime number of the form $4 * k + 3$, $c_1 = 0$ and $c_2 = (-1)^i$ (so the probe sequence is +0, -1, +4, -9, etc.) the probe sequence is a permutation. For example for $m = 7$ and $k = 3$:
  - $h(3, 0) = (3 \,\%\, 7 + 0^2) \,\%\, 7 = 3$
  - $h(3, 1) = (3 \,\%\, 7 - 1^2) \,\%\, 7 = 2$
  - $h(3, 2) = (3 \,\%\, 7 + 2^2) \,\%\, 7 = 0$
  - $h(3, 3) = (3 \,\%\, 7 - 3^2) \,\%\, 7 = 1$
  - $h(3, 4) = (3 \,\%\, 7 + 4^2) \,\%\, 7 = 5$
  - $h(3, 5) = (3 \,\%\, 7 - 5^2) \,\%\, 7 = 6$
  - $h(3, 6) = (3 \,\%\, 7 + 6^2) \,\%\, 7 = 4$

Open addressing - Quadratic probing - example

- Consider a hash table of size $m = 16$ that uses open addressing with quadratic probing for collision resolution (h'(k) is a hash function defined with the division method), $c_1 = c_2 = 0.5$.
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of m, c_1 and c_2.
  - *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
  - There are only m distinct probe sequences (once you have the starting position the whole sequence is fixed).

Open addressing - Double hashing

- In case of double hashing the hash function becomes:

$$h(k, i) = (h'(k) + i * h''(k)) \ \% \ m \ \forall i = 0, ..., m - 1$$

- where $h'(k)$ and $h''(k)$ are *simple* hash functions, where $h''(k)$ should never return the value 0.

- For a key, k, the first position examined will be $h'(k)$ and the other probed positions will be computed based on the second hash function, $h''(k)$.

Open addressing - Double hashing

- Similar to quadratic probing, not every combination of m and $h''(k)$ will return a complete permutation as a probe sequence.

- In order to produce a permutation m and all the values of $h''(k)$ have to be relatively primes. This can be achieved in two ways:
  - Choose m as a power of 2 and design h'' in such a way that it always returns an odd number.
  - Choose m as a prime number and design h'' in such a way that it always returns a value from the $\{0, m\text{-}1\}$ set (actually $\{1, m\text{-}1\}$ set, because $h''(k)$ should never return 0).

Open addressing - Double hashing

- Choose m as a prime number and design h'' in such a way that it always return a value from the {0, m-1} set.
- For example:
  $h'(k) = k\%m$
  $h''(k) = 1 + (k\%(m-1))$.
- For $m = 11$ and $k = 36$ we have:
  $h'(36) = 3$
  $h''(36) = 7$
- The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

Open addressing - Double hashing - example

- Consider a hash table of size $m = 17$ that uses open addressing with double hashing for collision resolution, with $h'(k) = k\%m$ and $h''(k) = (1 + (k\%16))$.
- Insert into the table the following elements: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39.
- Values of the two hash functions for each element:

key	75	12	109	43	22	18	55	81	92	27	13	16	39
h' (key)	7	12	7	9	5	1	4	13	7	10	13	16	5
h''(key)	12	13	14	12	7	3	8	2	13	12	14	1	8

Open addressing - Double hashing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

Open addressing - Double hashing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

- Main advantage of double hashing is that even if $h(k_1, 0) = h(k_2, 0)$ the probe sequences will be different if $k_1 \neq k_2$.
- For example:
  - 75: $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
  - 109: $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$
- Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Open addressing - operations

- In the following we will discuss the implementation of some 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 and compute the position for e again
    else
        ht.T[pos] ← e
   end-if
end-subalgorithm
```

Open addressing - other operations

- What should the *search* operation do?

Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?

Open addressing - other operations

- What should the *search* operation do?
- 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 - other operations

- What should the *search* operation do?
- 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 $\Theta(n)$

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.

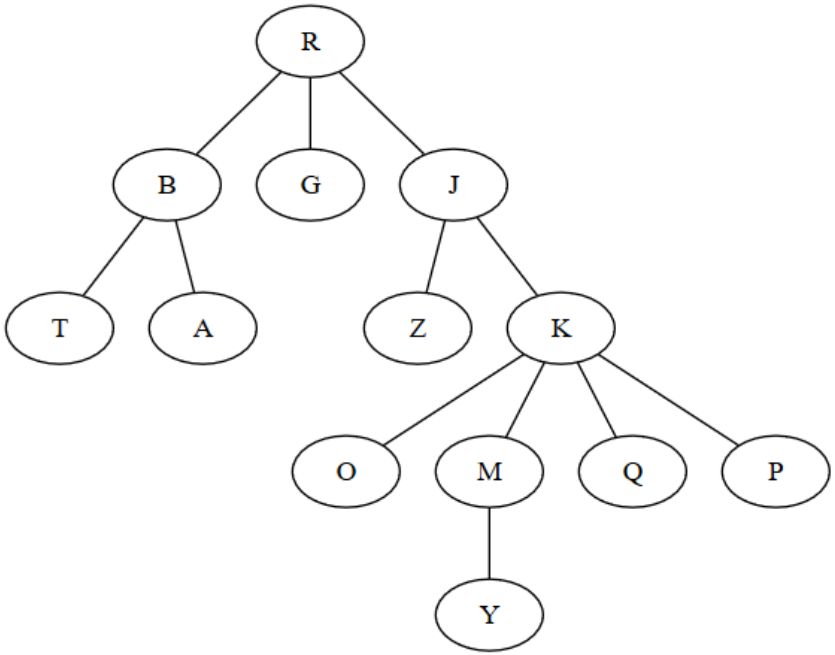
Tree - Terminology I

- 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
- Obs: this is doable if k is not too large

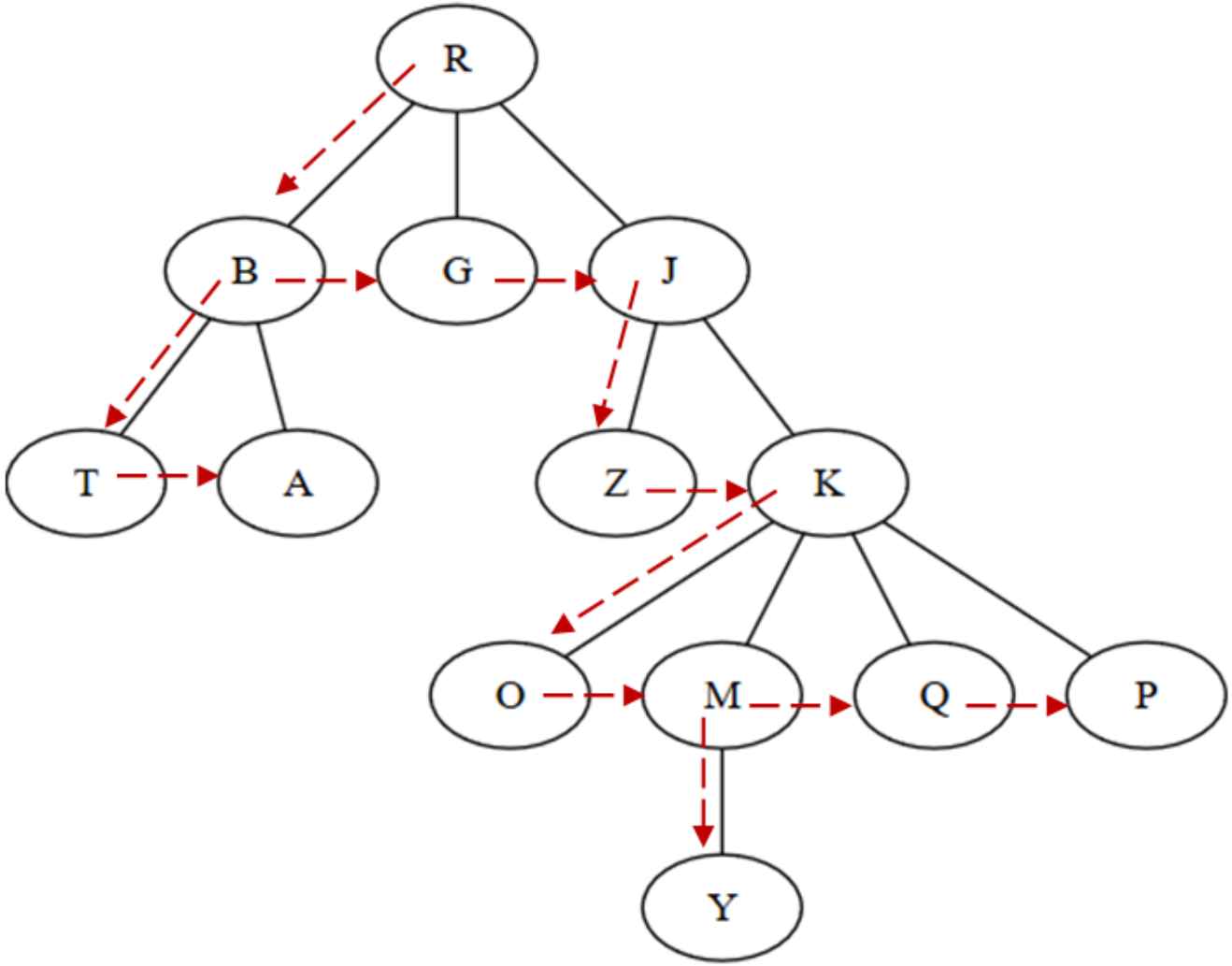
k-ary trees

- 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