CH-231-A

Algorithms and Data Structures

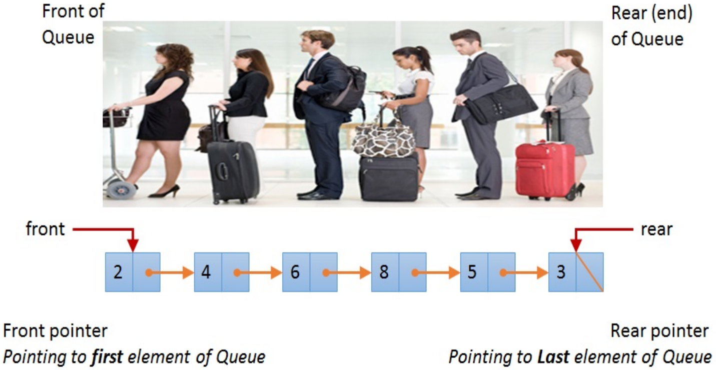
ADS

Lecture 20

Dr. Kinga Lipskoch

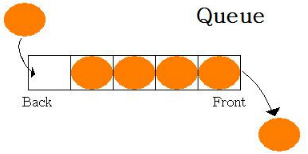
Spring 2020

Queue (1)


Front of Queue
Rear (end) of Queue
front
rear
2
4
6
8
5
3
Front pointer
*Pointing to **first** element of Queue*
Rear pointer
*Pointing to **Last** element of Queue*

Queue (2)

- Elementary dynamic data structure.
- Implements idea of dynamic set.
- Delete operation is called `dequeue`.
- Insert operation is called `enqueue`.
- FIFO principle (First In First Out):
  The element that is removed from the queue is the oldest one in the queue.



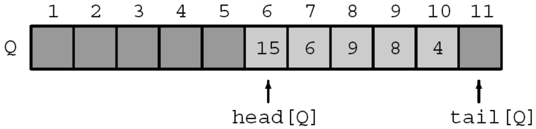
Queue Operations

Modify operations:

- *Enqueue*(Q, x):
  Add element x at the tail of queue Q.
- *Dequeue*(Q):
  If queue is non-empty, remove head element and return it.

Queue Example (Array Implementation) (1)

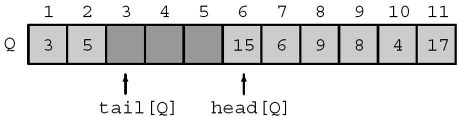
- *head*[Q] and *tail*[Q] mark the index of the first entry and the one following the last entry of the queue.
- Example:
  Queue with 5 elements between indices 6 (head) and 10 (tail).



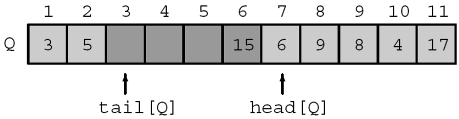
- We can also have under- and overflow.

Queue Example (Array Implementation) (2)

- Apply operations *Enqueue*$(Q, 17)$, *Enqueue*$(Q, 3)$, and *Enqueue*$(Q, 5)$:



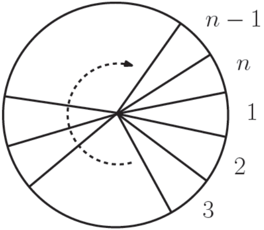
- Apply operation *Dequeue*(Q) returning entry 15:



Queue: Modulo Operations

Circular structure of filling the array with queue entries:

- $head[Q] = 1$ and $tail[Q] = 5$:
  4 entries
- $head[Q] = n - 1$ and $tail[Q] = 1$:
  2 entries
- $head[Q] = n$ and $tail[Q] = n - 1$:
  $n - 1$ entries (full queue)



Queue Operations (Array Implementation) (3)

```
Enqueue(Q,x)
if tail[Q] = head[Q]-1 then
  error 'overflow'
Q[tail[Q]] ← x
if tail[Q] = length[Q]
  then tail[Q] ← 1
  else tail[Q] ← tail[Q]+1
```

```
Dequeue(Q)
if tail[Q] = head[Q] then
  error 'underflow'
x ← Q[head[Q]]
if head[Q] = length[Q]
  then head[Q] ← 1
  else head[Q] ← head[Q]+1
return x
```

Queue Operations: Complexity

```
Enqueue(Q,x)
if tail[Q] = head[Q]-1 then
  error 'overflow'
Q[tail[Q]] ← x
if tail[Q] = length[Q]
  then tail[Q] ← 1
  else tail[Q] ← tail[Q]+1
```

```
Dequeue(Q)
if tail[Q] = head[Q] then
  error 'underflow'
x ← Q[head[Q]]
if head[Q] = length[Q]
  then head[Q] ← 1
  else head[Q] ← head[Q]+1
return x
```

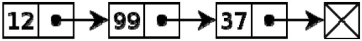
Complexity:
when implemented as an array all operations are $O(1)$.

Linked List (1)

- Another elementary dynamic data structure.
- Flexible implementation of idea of dynamic set.
- Implies a linear ordering of the elements.
- However, in contrast to an array, the order is not determined by indices but by links or pointers.
- The pointer supports the operations finding the succeeding (next) entry in the list.
- In contrast to arrays, lists do typically not support random access to entries.

Linked List (2)

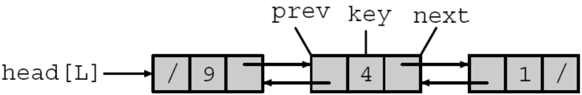
- Example of a linked list:



- Linked lists are dynamic data structures that allocate the requested memory when required.
- Start of linked list L is referred to as $head[L]$.
- $next[x]$ calls the pointer of element x and reports back the element to which the pointer of x is linking.

Doubly-Linked List

- A doubly-linked list enhances the linked list data structure by also storing pointers to the preceding (previous) element in the list.
- Hence, one can iterate in forward and backward direction.
- Example:



Linked List Operations

Queries:

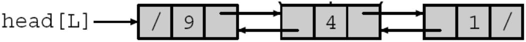
- Searching:

```
List-Search(L,k)
    x ← head[L]
    while x ≠ nil and key[x] ≠ k
        do x ← next[x]
    return x
```

- Time complexity: $O(n)$

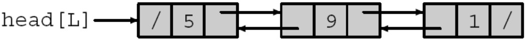
Modify Operations: Examples

- Example:

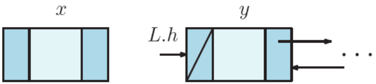


- Insert element x with $key[x] = 5$ (at beginning):

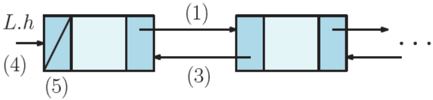
- Delete element x with $key[x] = 4$:



Insertion (at Beginning)




```
List-Insert(L, x)
    next[x] = head[L]
    if head[L] != nil
      then prev[head[L]] = x
    head[L] = x
    prev[x] = nil
```

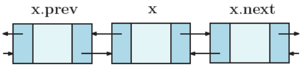


Time complexity: $\Theta(1)$

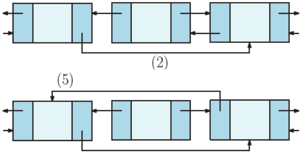
Insertion (Middle or End)

- We can also insert after a given element x.
- Time complexity:
  - $O(1)$, if element x is given by its pointer.
  - $O(n)$, if element x is given by its key (because of searching).

Deletion



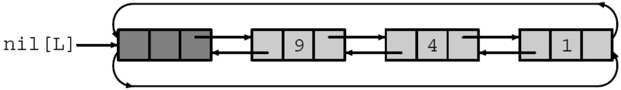

```
List-Delete(L,x)
    if prev[L] ≠ nil
       then next[prev[x]] ← next[x]
       else head[L] ← next[x]
    if next[x] ≠ nil
       then prev[next[x]] ← prev[x]
```



Time complexity:
$O(1)$ if we use pointer and $O(n)$ if we use key (because of searching).

Sentinels (1)

- In order to ease the handling of boundary cases, one can use dummy elements, so-called sentinels.
- Sentinels are handled like normal elements.
- One sentinel suffices when using circular lists.

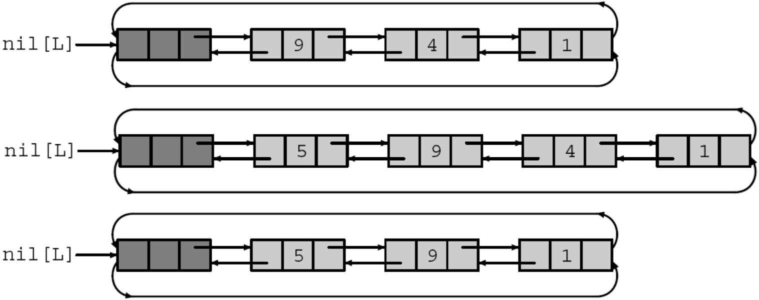



```
List-Search'(L,k)
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
    do x ← next[x]
return x
```

Sentinels (2)

```
List-Insert'(L,x)
    next[x] ← next[nil[L]]
    prev[next[nil[L]]] ← x
    next[nil[L]] ← x
    prev[x] ← nil[L]
```

```
List-Delete'(L,x)
    next[prev[x]] ← next[x]
    prev[next[x]] ← prev[x]
```

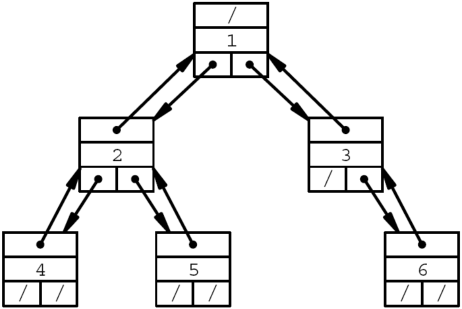


Representing Rooted Trees

- Traversing a rooted tree requires us to know about the hierarchical relationships of their nodes.
- Similar to linked list implementations, such relationships can be stored by using pointers.

Binary Tree

- Binary trees T have an attribute $T.root$.
- They consist of nodes x with attributes $x.parent$ (short $x.p$), $x.left$, and $x.right$ in addition to $x.key$.



d-ary Trees

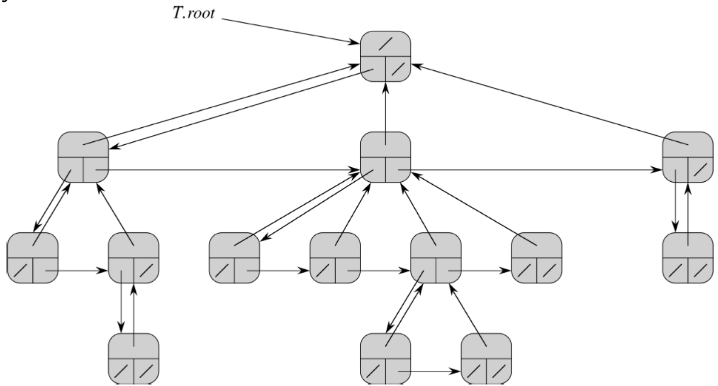
- d-ary trees are rooted trees with at most d children per node.
- They can be handled analogously to binary trees.

```
struct node {
    int val;
    node* parent;
    node* child[d];
};

typedef node* tree;
```

Rooted Trees with Arbitrary Branching

Rooted trees T with arbitrary branching consist of nodes x with attributes $x.p$, $x.leftmost\text{-}child$, and $x.right\text{-}sibling$ in addition to $x.key$.



Discussion

- Representing trees with pointers allows for a simple and intuitive representation.
- It also allows for a dynamic data management.
- Modifying operations can be implemented efficiently.
- However, extra memory requirements exist for storing the pointers.