Tirgul 5

Radix Sort
ADT
Trees Using Arrays

Radix Sort

- We have an input of n members in the range of $[1..k^d]$ for some k and d
- We can represent the input using d digits numbers, where each digit has k possible values
- We use d iterations of a stable linear sorting (counting sort) for sorting the input, starting with the least significant digit
- The total running time is $\Theta(d \cdot (n+k))$

Radix Sort

- Proof of correctness:
- We prove that the algorithm is correct by induction.
  - The induction hypothesis is that after i steps, the numbers are sorted by i less significant digits.
  - Inductively, step i+1 sorts by the $(i+1)^{th}$ least significant digit.
    - If two numbers have the same $(i+1)^{th}$ least significant digit, the stability property of counting sort leaves them sorted by lower order digits
    - and if they don't have the same value, the counting sort on step i+1 puts them in the right order, so in either case the induction hypothesis holds.

Radix Sort

- Note that when $d > \log(n)$, $\Theta(d \cdot (n+k)) > \Theta(n \cdot \log(n))$ and we will prefer merge-sort
- When k is small, we may simply prefer using counting-sort

Summary:

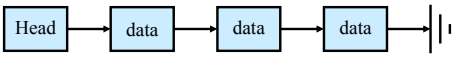
- The decision of which sorting algorithm to use should be taken after considering the size and range of the input (and other factors we didn't mention, like space complexity…)

ADT

- An *Abstract Data Type*, or *ADT*, is the specification of a set of data and a set of operations that can be performed on the data.
- It is abstract in the sense that the actual implementation is not defined and does not affect the behavior of the *ADT*.
- In a computer programs (Java), the *ADT* is represented by an interface, which shields the implementation details.

List ADT

- So far we have not separated the list interface from its implementation



- In fact the list is an *ADT* that can be implemented in various ways

List ADT

- Our definition of the list *ADT* is therefore:
  - A data structure that may hold any number of elements
  - We want to allow the following operations:
    - insert a new element
    - Find an element
    - Remove an element
    - Iterate over all elements
  - Other optional operations are:
    - Sort the elements
    - Insert in-place
    - etc.

List ADT

- For example, we may implement the list *ADT* using an array:

0 1 2 3 4 5 6 7 8 9

or using a double array:

D	orange	apple	banana	olive	date	Pome-granate	fig	NULL	NULL	NULL
N	2	5	4	1	7	6	3			
	0	1	2	3	4	5	6	7	8	9

Queue ADT

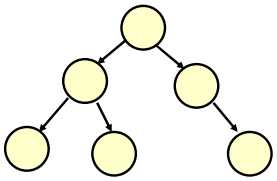
- A queue is a first-in-first-out (*FIFO*) sequential data structure in which elements are added (*enqueued*) at one end and are removed (*dequeued*) from the other end
- In the queue *ADT* we have the following operations:
  - initialization
  - *enqueue*– add an element to the end of the queue
  - *dequeue*– remove an element from the head of the queue
  - (optionally) count the number of elements in the queue
  - (optionally) view the first element in the queue

Stack ADT

- Another useful *ADT* is the stack, a last-in-first-out (*LIFO*) sequential data structure in which elements are added (*pushed*) at one end and are removed (*popped*) from the same end
- In the stack *ADT* we have the following operations:
  - Initialization
  - *push* - add an element to the top of the stack
  - *pop* – remove an element from the top of the stack
  - (optionally) count the number of elements in the stack
  - (optionally) *top* - view the top element in the stack

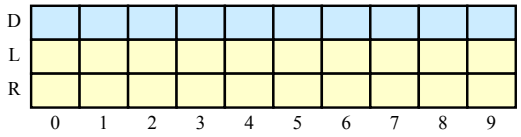
Trees Using Arrays

- So far our (binary) trees looked like this:



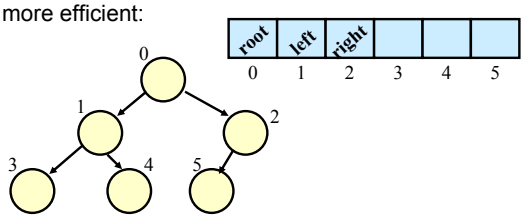
Trees Using Arrays

- Trees, like lists, can be implemented using arrays
- In the general case we will have something like this:



Trees Using Arrays

- Many times we want our trees to be complete
- For a complete tree, the implementation is much more efficient:



Trees Using Arrays

- Each node has (up to) 2 children, left and right
- If the tree is complete, we can position the left child of node ***i*** at 2***i***+1 and its right child at 2***i***+2
- The parent of node ***i*** is located at floor((***i*** -1)/2)
- We don't need pointers, we can access each element in O(1)
- What happens if the tree is not complete?