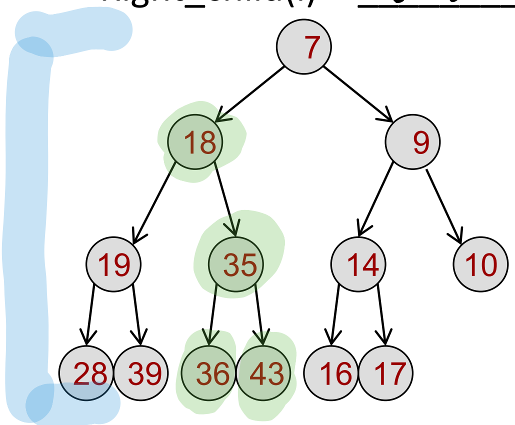
Array-based and Link-based

TREE IMPLEMENTATIONS

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = i / 2
  - Left_child(i) = i * 2
  - Right_child(i) = i * 2 + 1

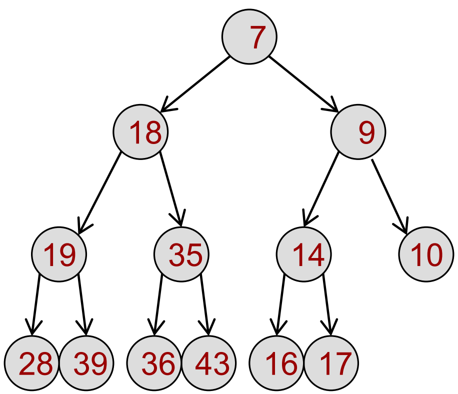


0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = 2
Left_child(5) = 10
Right_child(5) = 11

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding node i's parent, left, and right child?
  - Parent(i) = i/2
  - Left_child(i) = 2*i
  - Right_child(i) = 2*i + 1



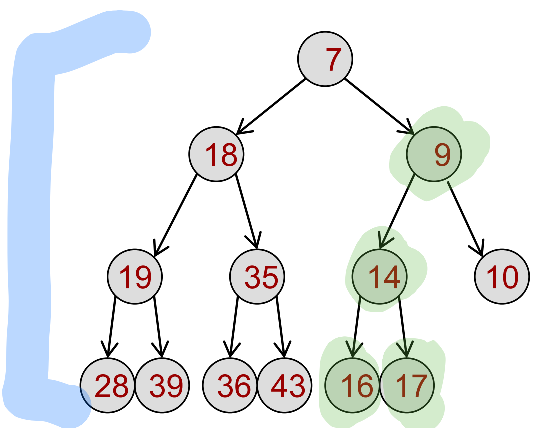
0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = 5/2 = 2
Left_child(5) = 2*5 = 10
Right_child(5) = 2*5+1 = 11

Non-complete binary trees require much more bookkeeping to store in arrays…usually link-based approaches are preferred

0-Based Indexing

- Now let's assume we start the root at index 0 of the array
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = $(i-1)/2$
  - Left_child(i) = $2i+1$
  - Right_child(i) = $2i+2$



0	1	2	3	4	5	6	7	8	9	10	11	12
7	18	9	19	35	14	10	28	39	36	43	16	17

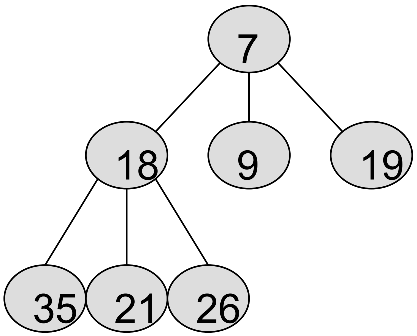
parent(5) = 2

Left_child(5) = 11

Right_child(5) = 12

D-ary Array-based Implementations

- Arrays can be used to store d-ary **complete** trees
  - Adjust the formulas derived for binary trees in previous slides in terms of **d**



A 3-ary (trinary) tree

0	1	2	3	4	5	6
7	18	9	19	35	21	26

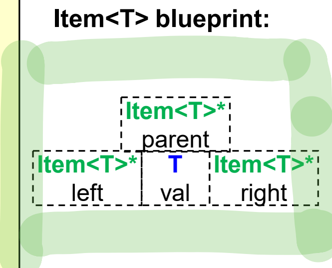
As an exercise
how can you generalize formulas ?
for any d-ary trees.

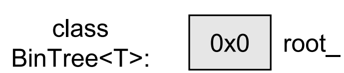
Link-Based Approaches

- For an arbitrary (**non-complete**) d-ary tree we need to use pointer-based structures
  - Much like a linked list but now with two pointers per Item
- Use NULL pointers to indicate no child
- Dynamically allocate and free items when you add/remove them

```
#include<iostream>
using namespace std;

template <typename T>
struct Item {
  T val;
  BTItem<T>* left,right;
  BTItem<T>* parent;
};
// Bin. Search Tree
template <typename T>
class BinTree
{
 public:
 BinTree();
 ~BinTree();
 void add(const T& v);
 ...
 private:
 Item<T>* root_;   // pointer to
                   // root
};
```

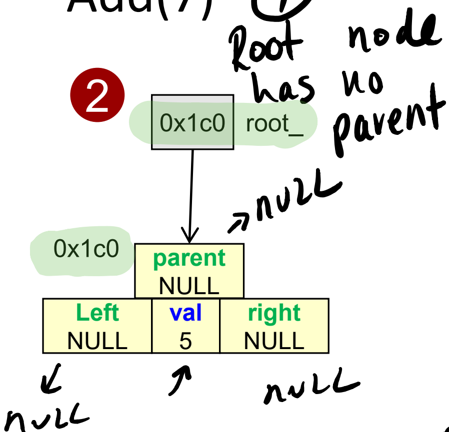


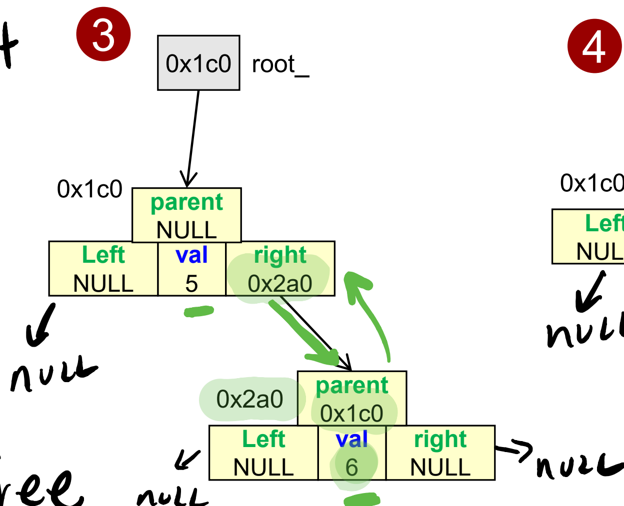


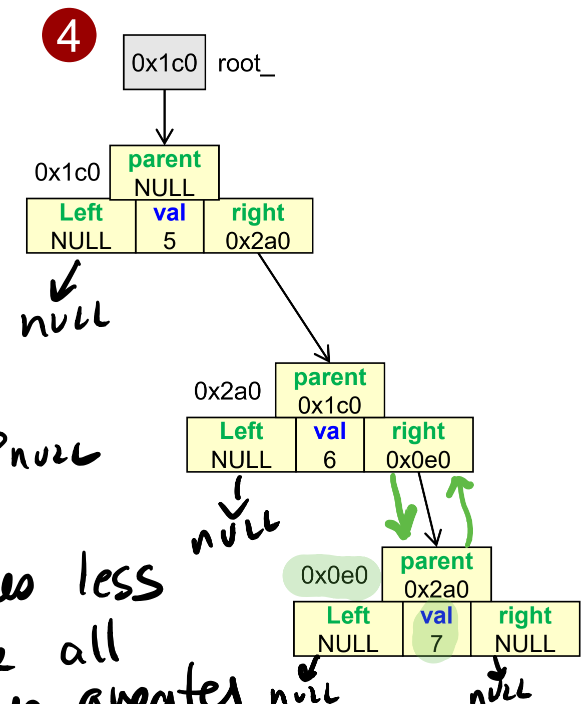
Link-Based Approaches

- Add(5) 2
- Add(6) 3
- Add(7) 4

1 class LinkedBST: 0x0 root_ = NULL
// can be
// done in constructor







Binary Search Tree property: key values are st left subtree has all key values less than node and right subtree all key values greater