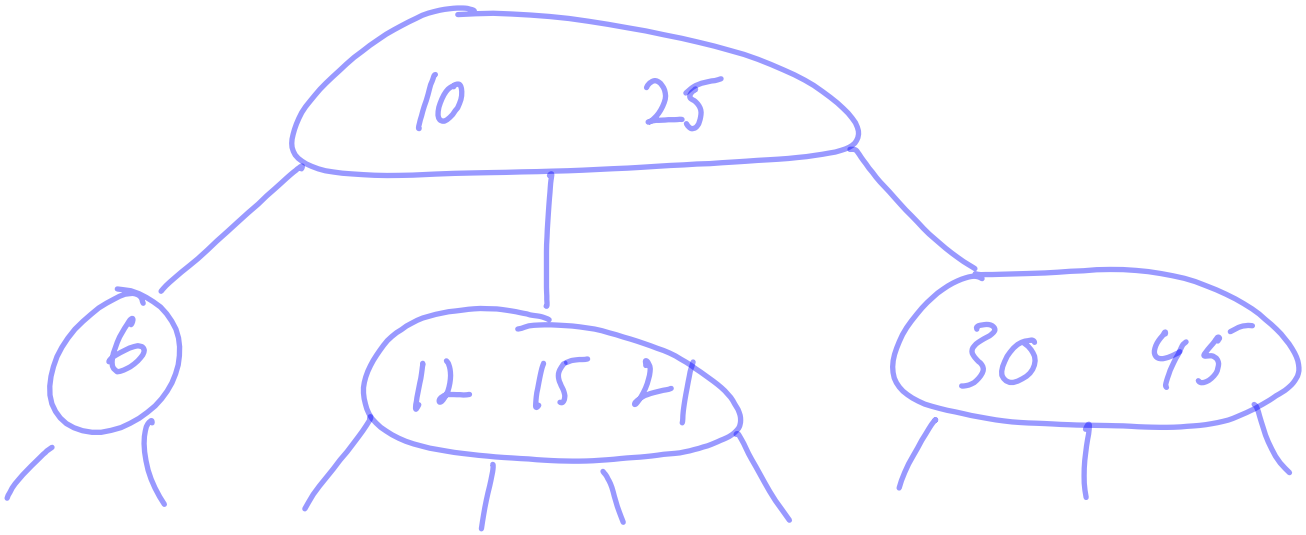
Last time we considered Red-Black trees which are a special type of *binary* search tree.

We can generalize the notion of binary search tree to a **k-ary search tree**. That is, a search tree in which a node may have up to k children (a binary tree is a 2-ary tree).

Suppose a node has $k' \leq k$ children, and we are looking for some value x. x is not in the current node, and we want to determine which child to go to to find x.

The node will store $k' - 1$ sorted values in this node (as opposed to just 1 in the case of binary trees). These values will help us find which child to search.

Example of a 4-ary tree:



Red-Black Trees allow for search, insert, and delete to be done in $O(\log n)$ time in the worst case. Why would we need to consider a k-ary tree?

Suppose we are dealing with a large amount of data which will not fit into main memory. Perhaps this data is being stored in secondary storage, and each read (or write) from this storage is expensive (where as we assumed we could read a value in a node in a red-black tree in constant time).

In this setting, we assume can read a block of data with the same cost as reading just one element. Having more values stored in a node of our tree allows us to read many values in at once, cutting down on the total number of disk reads we need to make.

We are interested in considering a k-ary search tree which allows us to search, insert, and delete from the tree as efficiently as possible.

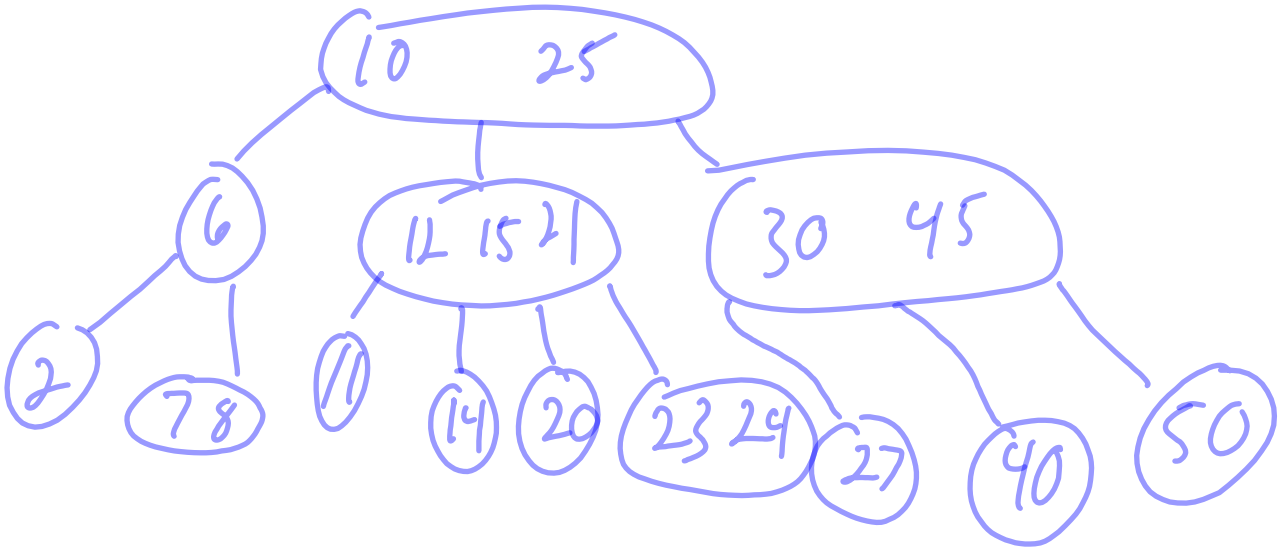
Again, if we get an "unlucky" sequence of inserts, the tree could become "unbalanced" which will hurt the worst case performance of the algorithm.

min degree = min # of children.

One such tree is a **B-tree**. A B-tree T with *minimum degree* $t \geq 2$ is defined as follows:

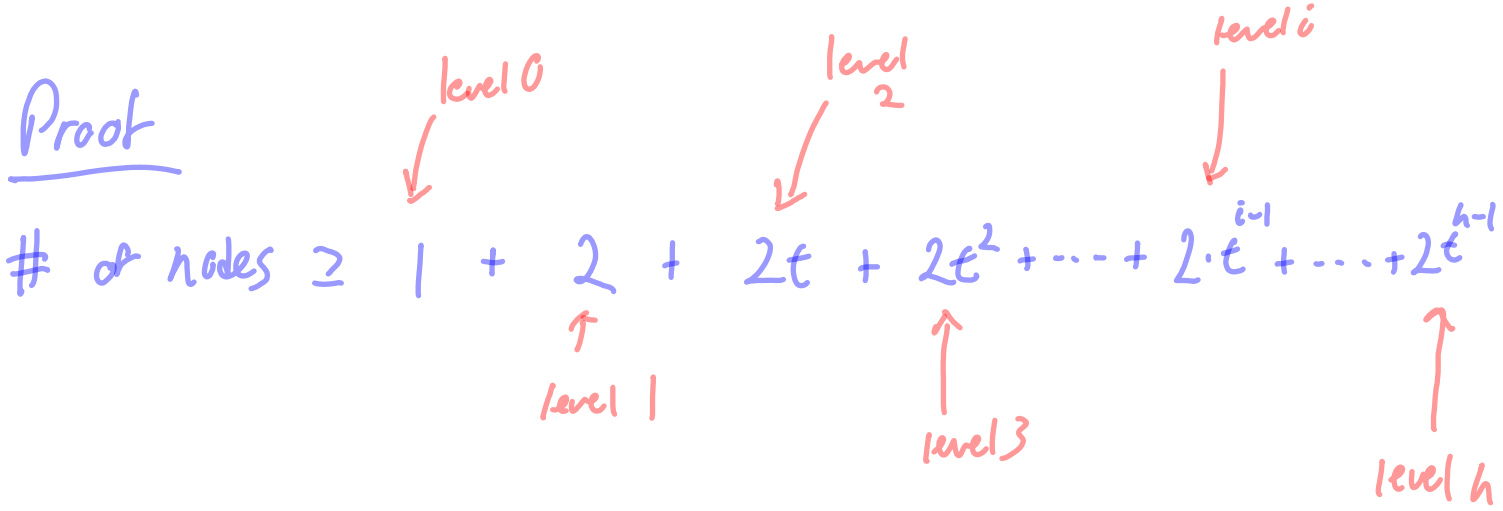
1. T is a $2t$-ary search tree (each node has at most $2t$ children and stores at most $2t - 1$ values).
2. Every node, except the root, stores at least $t - 1$ values (the root must store at least 1 value).
3. All leaves are at the same depth.

Example: $t=2 \Rightarrow$ 4-ary search tree



Theorem: A B-tree with minimum degree $t \geq 2$ which stores n values has height $h \leq \log_t \frac{n+1}{2}$.

Proof



$$= 1 + \sum_{i=1}^{h} 2t^{i-1} = 1 + 2\sum_{i=0}^{h-1} t^i$$

$$= 1 + 2\left[\frac{t^h - 1}{t - 1}\right]$$

$$\#\text{ of values} = n \geq 1 + \cancel{(t-1)} \cdot 2\left[\frac{t^h - 1}{\cancel{t-1}}\right]$$

$$= 1 + 2t^h - 2 = 2t^h - 1$$

$$\Rightarrow n \geq 2t^h - 1 \Rightarrow t^h \leq \frac{n+1}{2} \Rightarrow \log_t t^h \leq \log_t \frac{n+1}{2}$$

$$\Rightarrow \boxed{h \leq \log_t \frac{n+1}{2}}$$

Recursive algorithm for searching for the value key in a B-tree T. Initial call: B-Tree-Search($T.root, key$).

B-Tree-Search(x,key)
  $i = 1$
  while $i \leq$ number of keys in x AND $key > i$th key in x **do**
    $i = i + 1$
  end while
  if $i \leq$ number of keys in x AND $key ==$ ith key in x **then**
    return ith key in x
  else
    if x is a leaf **then**
      return NIL
    end if
  end if
  $b =$ DISK-READ(ith child of x)
  return B-Tree-Search(b, key)

O(t)

O(1)

At most $\log_t \frac{n+1}{2}$ recursive calls.

$\Rightarrow$ CPU Running time: $O(t \cdot \log_t n)$

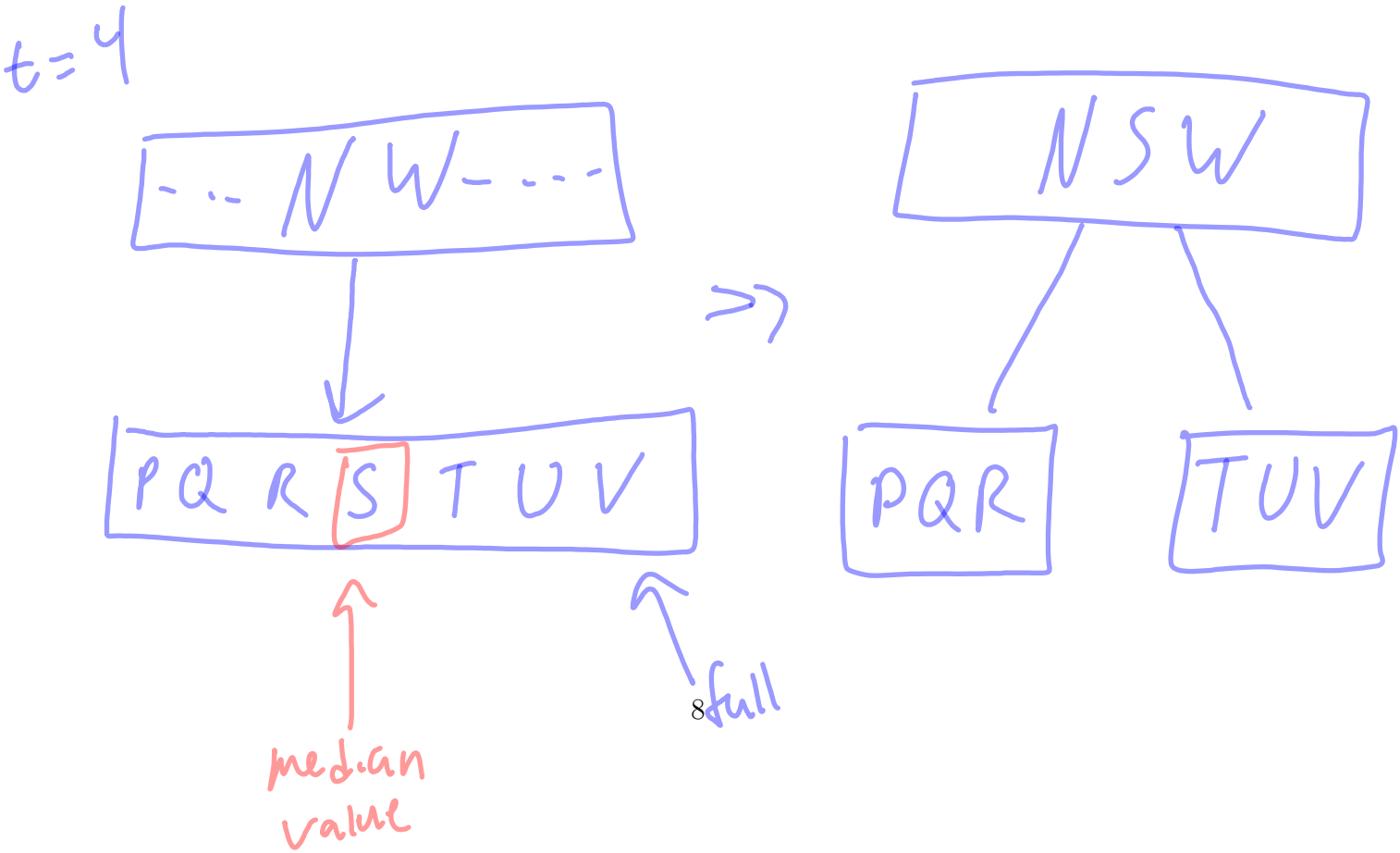
How many calls to DISK-READ?

$O(\log_t n)$

Inserting is different than typical inserts into a search tree because we require that the leaves always be at the same level.

Intuition: nodes are only allowed to hold at most $2t-1$ values. We can insert the node into an existing leaf as long as it isn't "full". That is, if it currently has fewer than $2t-1$ values.

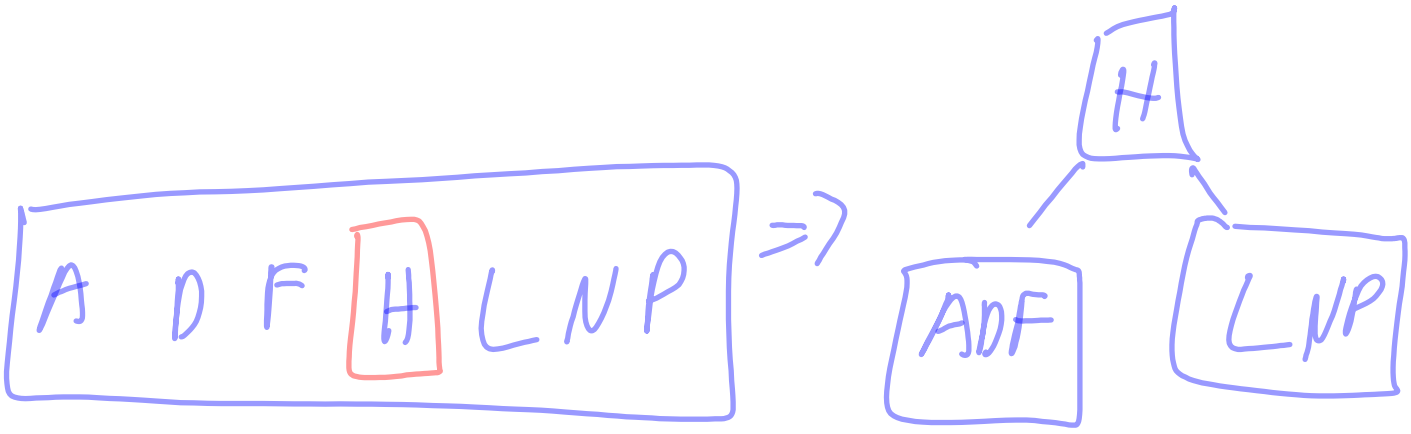
If it is full, we can split a node into two new nodes in the following way:



This technique only works if the parent node also is not full (inserting the median value into the parent would give the parent too many values).

We can make sure this doesn't happen by splitting any full node we find as we search for the proper location to insert the new value.

What if the root is full?



Running time of insert:

1. $O(t)$ runtime per node.
2. The path has height $O(\log_t n)$.
3. The total CPU running time is $O(t \log_t n)$.

How many reads/writes to the secondary storage do we make?

2 writes per split $\leq O(\log_t n)$ splits

One read per level. $\leq O(\log_t n)$ reads.

Delete is similar but slightly different. It has the same worst case complexity.

No rotations are needed. Splitting (or merging in the case of delete) maintains the balance of the tree.

The height of the tree grows (or shrinks) in height only when splitting (or merging) the root.