INTRODUCTION

This week we cover the following subjects

8.1 Trees

8.1.1	Why Trees?
8.1.2	Terminology
8.1.3	Binary Trees
8.1.4	Search Example
8.1.5	Adding a Node
8.1.6	Deleting a Node
8.1.7	`bintree.h`
8.1.8	Maintenance
8.1.9	Performance
8.1.10	Other Schemes

8.1 TREES

- Lists are very useful, but have limitations
- This lecture, introduce *trees*
  - Concepts
  - Introduce implementation

8.1.1 Why Trees?

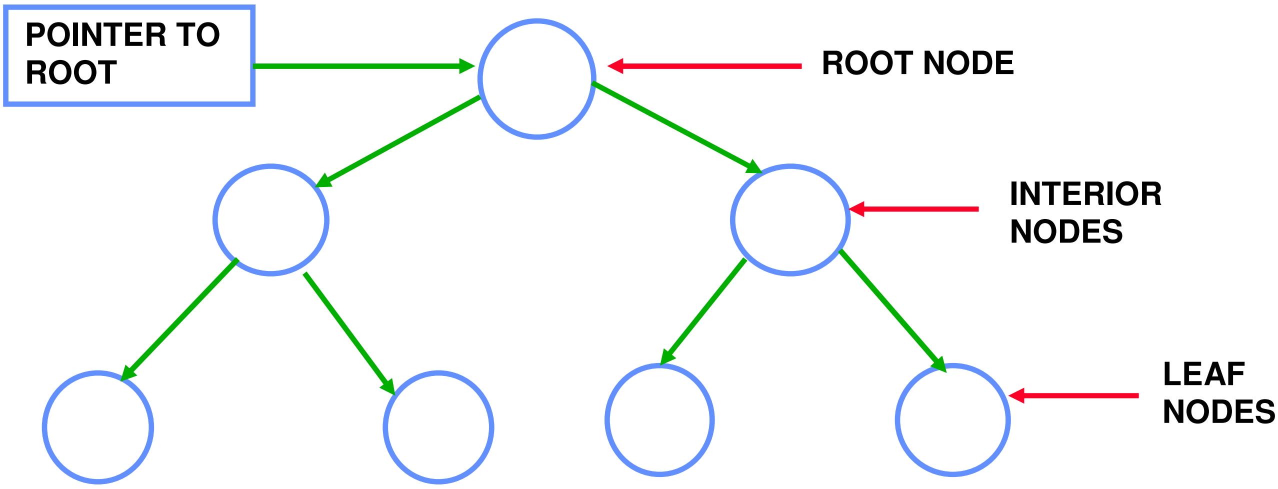
- The list data structure can be used for any data processing task.

- But isn't always the best choice:
  - Simple arrays are better for many problems
  - Lists are slow for random access to large data volumes.

- Consider a small database of 100,000 items
  - The only way to search a linked list is by starting from one end
  - Average search time proportional to $n/2$, so 50,000 in this case
  - Trees provide more complex but more efficient searching

8.1.2 Terminology

- Basic decisions:
  - How many children can an interior node have?
  - Do interior nodes contain data, or search information only?



8.1.3 Binary Trees

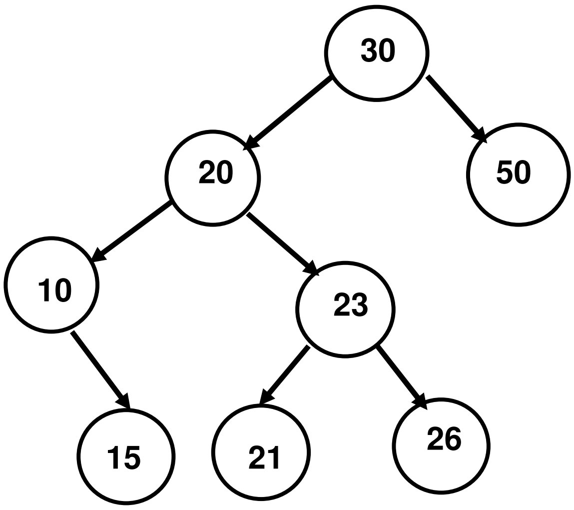
- Each node is a structure that points to at most two other nodes and contains a key value plus any data we want to store in the tree that is associated with the key.

- Key need not be an integer; used here for simplicity.

- Every node has the following properties. All nodes in the left sub-branch will have a key value less than the one in the node. All nodes in the right sub-branch will have a key value greater than the one in node.

8.1.4 Search Example

- Consider the tree below, search for keys 21, 40



- Basic operation: search (add, delete, etc, require being able to search first)

- Search is *recursive*: If the key is less than the node then search the left sub-tree. If the key equals the node key then search is found. If the key is greater then the node key then search the right sub-branch.

8.1.5 Adding A Node

- To add a node we search through the tree looking for the correct node to add the new node too.

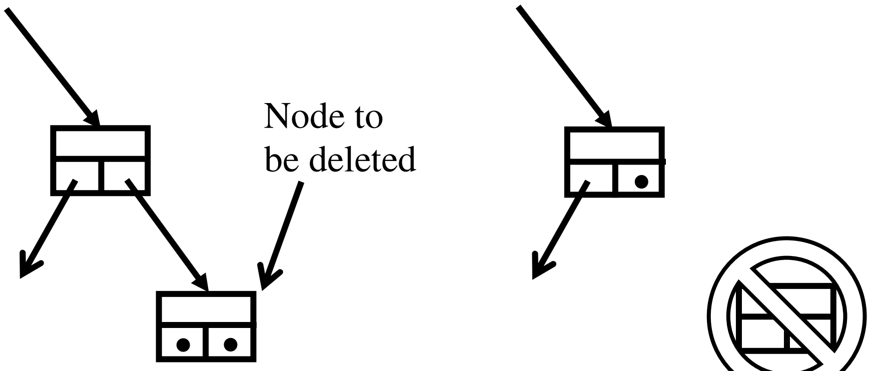
- The new node will become a new leaf node in the tree.

8.1.6 Deleting A Node

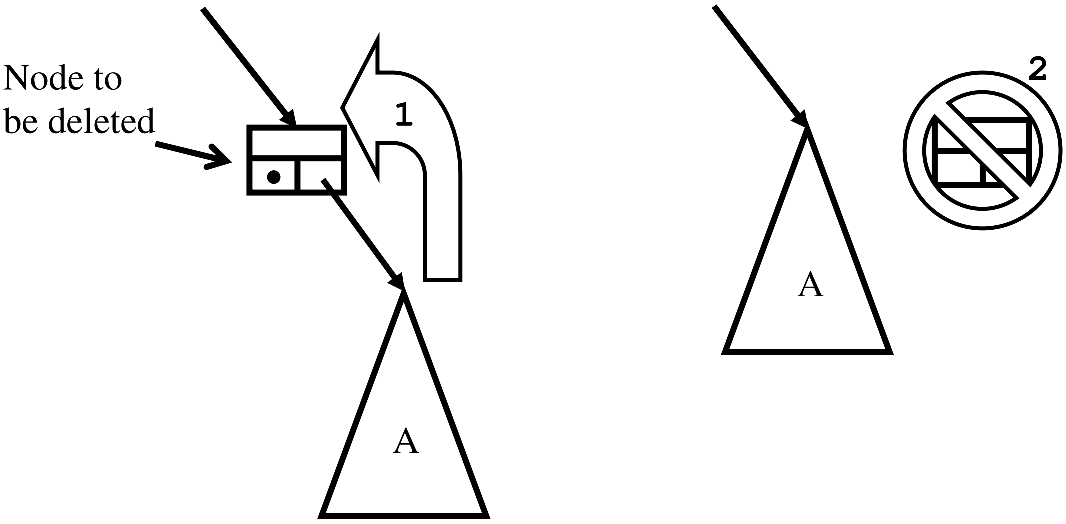
- While adding nodes to a tree is fairly easy the same cannot be said for deleting nodes.

- The problem lies with the root and interior nodes, which may have two sub-trees hanging off them.

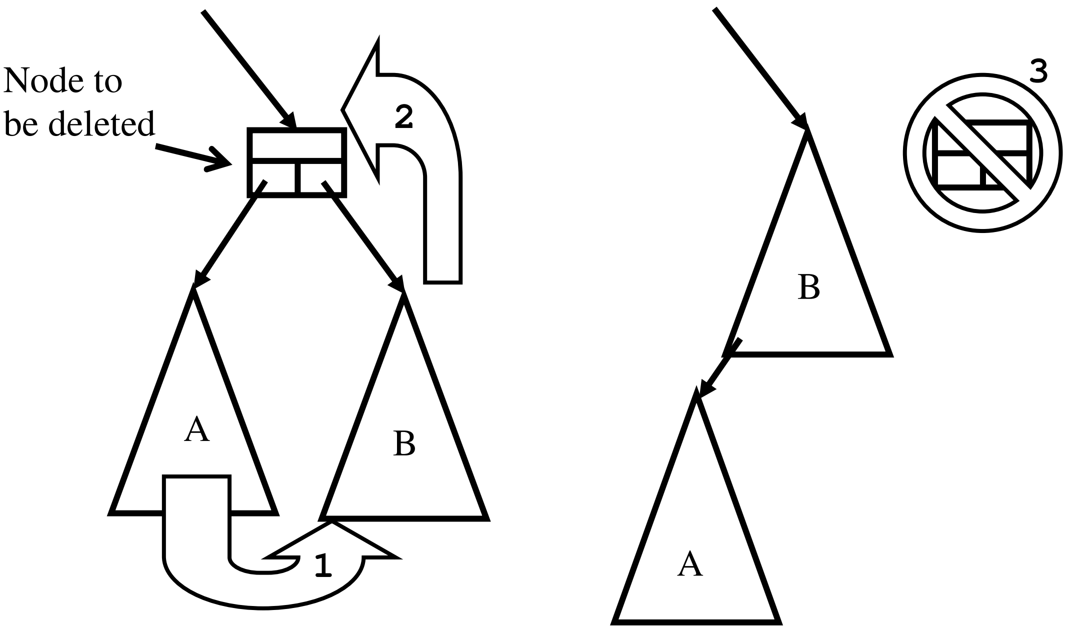
Deleting a leaf node

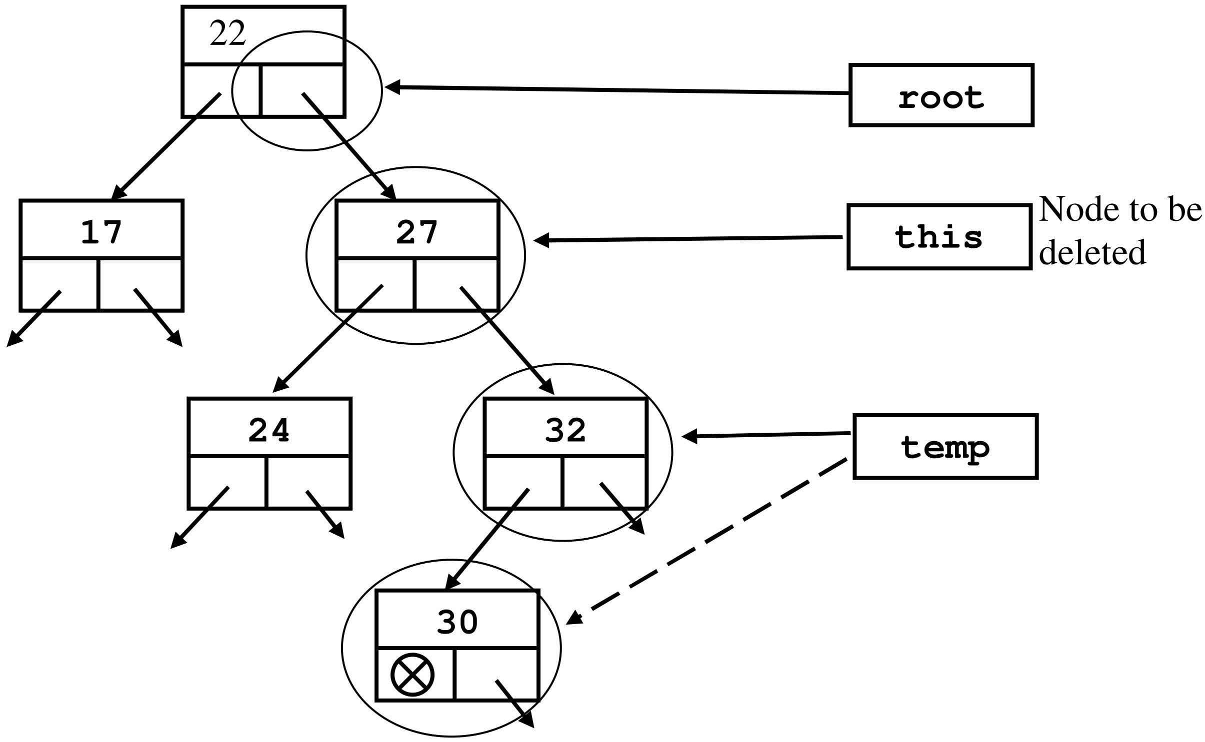


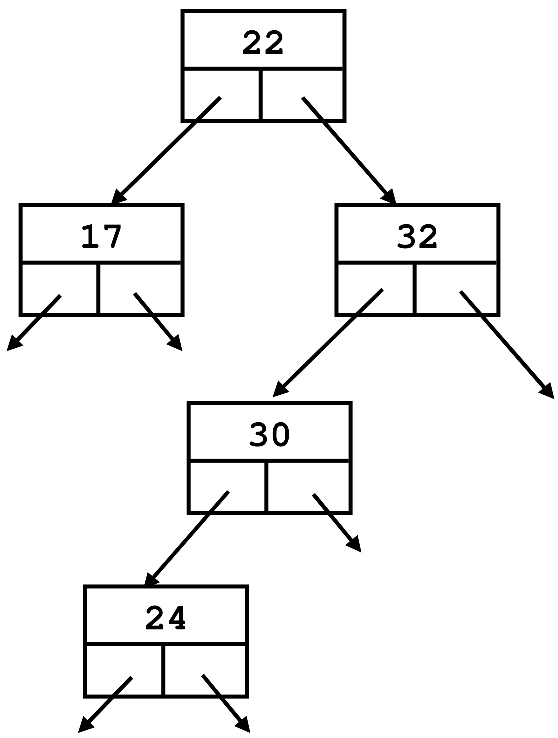
Deleting an interior node with only one sub-tree filled.



Deleting an interior node with both sub-trees filled.




22
root
17
27
this
Node to be deleted
24
32
temp
30


22
17
32
30
24

8.1.7 **bintree.h**

```
sally% cat binide.h

#ifndef BINNODE_H
#define BINNODE_H

/*********************************************\
   template node class for binary tree
\*********************************************/

template <typename dataType> class binNode
{
   private:
      dataType nodeData;
      binNode<dataType> *left, *right;

      void deleteNode(binNode<dataType> **root) {

         if (left == NULL && right == NULL) {
            // leaf node
            *root = NULL;
         } else if (left == NULL) {
            // right branch but no left branch
            *root = right;
         } else if (right == NULL) {
            // left branch but no right branch
            *root = left;
         } else {
            // has left and right branch
            binNode<dataType> *temp = right;

            // find left most node of right
            // branch
            while (temp->left != NULL)
                temp = temp->left;

            // attach left branch to left side of
            // right branch
            temp->left = left;
```

```
            // make root point to right branch
            *root = right;
         }
         delete (this);
      }

   public:
      // constructors
      binNode() : left(NULL), right(NULL) {}

      binNode(const dataType& dataItem) :
         nodeData(dataItem),
         left(NULL), right(NULL) {
      }

      // destructor
      ~binNode() {
         if (left != NULL) delete left;
         if (right != NULL) delete right;
      }

      void insert(const dataType& dataItem) {

         if (nodeData == dataItem) {
            throw std::invalid_argument(
               "dataItem already in tree");
         }

         if (dataItem < nodeData) {
            if (left == NULL) {
               left = new binNode(dataItem);
            } else {
               left->insert(dataItem);
            }
         } else {
            if (right == NULL) {
               right = new binNode(dataItem);
            } else {
               right->insert(dataItem);
            }
         }
      }
```

```
void erase(binNode<dataType> **root,
           const dataType &delData) {

   if (delData == nodeData) {
      deleteNode(root);
   } else {
      if (delData < nodeData) {
         if (left == NULL) {
            throw std::invalid_argument(
                 "delItem not in tree");
         } else {
            left->erase(&left, delData);
         }
      } else {
         if (right == NULL) {
            throw std::invalid_argument(
                "delItem not in tree");
         } else {
            right->erase(&right, delData);
         }
      }
   }
}

bool findData(const dataType &data,
              dataType &found) {
   if (data == nodeData) {
      found = nodeData;
      return true;
   } else if (data < nodeData) {
      if (left == NULL) return false;
      else return
           left->findData(data, found);
   } else {
      if (right == NULL) return false;
      else return
           right->findData(data, found);
   }
}

// overloaded dereference operator
const dataType& operator * () const {
   return nodeData;
}
```

```
};

#endif

// ##############################################

sally% cat bintree.h

#ifndef BINTREE_H_
#define BINTREE_H_

#include <stdexcept>

#include "binnode.h"

/***********************************************\
   template class for a binary tree
\***********************************************/

template <typename dataType> class bintree
{
   private:
      binNode<dataType> *root;
      int numItems;

   public:
      /*****************************************\
         constructors & destructors
      \*****************************************/

      // constructor
      bintree() : root(NULL), numItems(0) {}

      // destructor
      ~bintree() {
         if (root != NULL) delete root;
      }

      /*****************************************\
         misc functions
      \*****************************************/
```

```
      bool empty() const {
         return (root == NULL);
      }

      int size() const {
         return numItems;
      }

      /****************************************\
         insertion and erasure functions
      \****************************************/

      void insert(const dataType& newData) {
         if (root == NULL) {
            root =
               new binNode<dataType>(newData);
         } else {
            root->insert(newData);
         }
         numItems++;
      }

      void erase(const dataType& delData) {

         if (root == NULL) {
            throw std::invalid_argument("data
               does not exist in tree to erase");
         }

         root->erase(&root, delData);

         numItems--;
      }

      bool findData(const dataType &data,
                    dataType &found) {
         if (root == NULL) return false;
         else return root->findData(data, found);
      }
};

#endif

// ################################################
```

```
sally% cat testmain.cpp

/**********************************************\
   Test program for demonstrating
   container types
\**********************************************/
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <algorithm>

#include "dataobject.h"
#include "bintree.h"

using namespace std;

double difUtime(struct timeval *first,
                struct timeval *second);

double difUtime(struct timeval *first,
                struct timeval *second)
{
   // return the difference in seconds,
   //  including milli seconds

   double difsec =
      second->tv_sec - first->tv_sec;
   double udifse =
      second->tv_usec - first->tv_usec;

   return (difsec + udifsec / 1000000.0);
}

int main()
{
   const int MAXDATA = 1000000;
   dataObject *doPtr, data;
   int i, keyvals[MAXDATA];
   bintree<dataObject> testContainer;
```

```
// data for calculating timing
struct timeval first, second;
double usecs;

try {
   /****************************************\
      Initialise things to demonstrate the
      container
      - fill keyvals and scramble it
   \****************************************/

   for (i=0; i<MAXDATA; i++) keyvals[i] = i;
   srand(time(NULL));
   for (i=0; i<MAXDATA; i++)
      swap(keyvals[i],
           keyvals[random() % MAXDATA]);
   int middle = keyvals[MAXDATA/2];

   /****************************************\
      test inserting MAXDATA data pieces into
      the container with keyval 0 to
      MAXDATA-1 in random order
   \****************************************/

   gettimeofday(&first, NULL);
   for (i=0; i<MAXDATA; i++) {
      doPtr = new dataObject(keyvals[i]);
      testContainer.insert(*doPtr);
   }
   cout << "\n";
   gettimeofday(&second, NULL);
   usecs = difUtime(&first, &second);
   cout << MAXDATA << " items in container in
              random order\n";
   cout << "time taken to push data into
     container = " << usecs << " seconds\n\n";

   /****************************************\
      test finding data in the container
   \****************************************/

   gettimeofday(&first, NULL);
   testContainer.findData(
              dataObject(0), data);
```

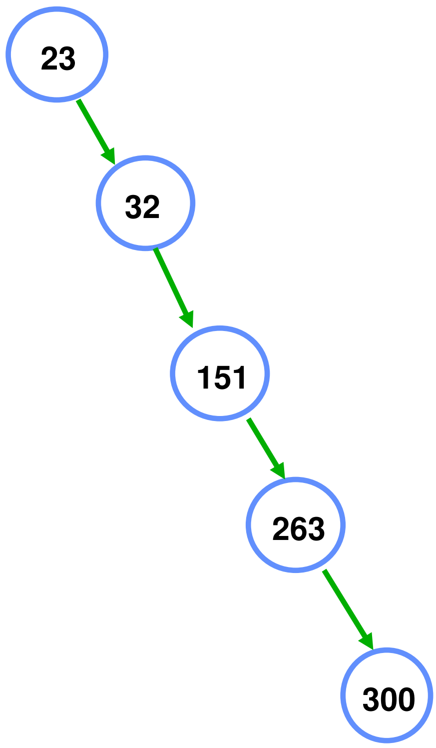
```
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find first item in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.findData(
       dataObject(keyvals[middle]), data);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find item in middle
 of container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.findData(
              dataObject(MAXDATA), data);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to find item doesn't
   exit in container = " << usecs <<
    "seconds\n\n";

/****************************************\
   test removing data from the container
\****************************************/

gettimeofday(&first, NULL);
testContainer.erase(dataObject(0));
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase first item
 in container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
dataObject temp(keyvals[middle]);
testContainer.erase(temp);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "time taken to erase middle item in
    container = " << usecs << " seconds\n";

gettimeofday(&first, NULL);
testContainer.erase(dataObject(MAXDATA-1));
gettimeofday(&second, NULL);
```

```
      usecs = difUtime(&first, &second);
      cout << "time taken to erase last item in
          container = " << usecs << " seconds\n";
   } catch (out_of_range &ex) {
      cout << "\nERROR - Out of Range Exception
            thrown\n" << ex.what() << "\n";
      exit(1);
   } catch (invalid_argument &ex) {
      cout << "\nERROR - Invalid Argument
        Exception thrown\n" << ex.what() << "\n";
      exit(1);
   } catch(...) {
      cout << "\nERROR - undefined Exception
          thrown\n";
      exit(1);
   }

   return 0;
}

// ##############################################

sally %cat makefile

CC = g++
prog: testmain.o
     $(CC) testmain.o -Wall -o testmain
testmain.o: testmain.cpp bintree.h dataobject.h
     $(CC) -Wall -c testmain.cpp

sally% testmain

1000000 items in container in random order
time taken to push data into container = 28.5358
seconds

time taken to find first item in container =
2.6e-05 seconds
time taken to find item in middle of container =
3.7e-05 seconds
time taken to find item doesn't exit in container
= 2e-05 seconds
```

```
time taken to erase first item in container = 6e-
05 seconds
time taken to erase middle item in container =
3.5e-05 seconds
time taken to erase last item in container =
1.8e-05 seconds
```

8.1.8 Maintenance

- Tree performs best if it is *balanced.*
- If we just add entries as they arrive, tree can get very unbalanced.
- Thus, tree needs maintenance.
- Code can be quite complex.
- Tree can be re-organised with each addition.
- Or do it "off-line" occasionally.



8.1.9 Performance

- If tree is properly balanced, then for n entries, average no. of checks = $\lg_2(n) - 1$. Therefore $O(\lg_2 n)$

- But for linear search, average = $n / 2$.

- Thus, if n = 100,000, checks = 17 for balanced tree, compared to 50,000 for linear search

- $\lg_2(n) - 1$ is a *lower bound.*

- Bound is approached, even if tree is somewhat unbalanced.

8.1.10 Other Schemes

- Binary tree with embedded data is well understood:
  - Lots of theoretical analysis
  - Popular as an introductory data structure

- But other approaches have been developed:
  - > 2 children per interior node
  - Data only in leaf nodes; interior nodes contain only search information (index values and pointers)
  - Thus, different structures for interior nodes and leaf nodes
  - Leaf nodes might be records on disk, interior nodes in memory

- More branching per node gives shorter searches:
  - for k entries per node, average hits = $\lg_k(n)$
  - For $k = 5$, $n = 100{,}000$, average hits = 7