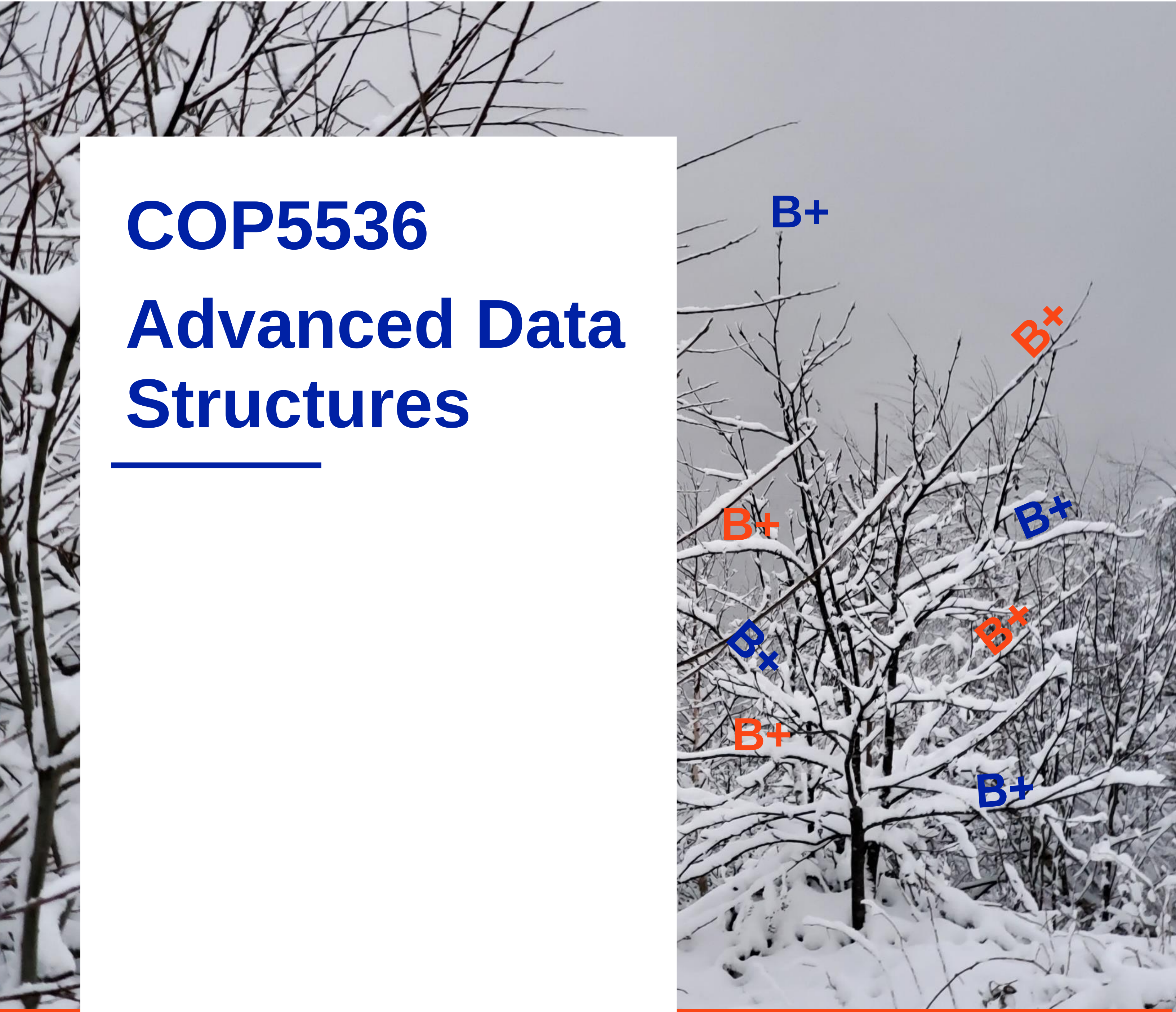
COP5536 Advanced Data Structures



APRIL 6

Spring 2021 – Programming Project
Katarina Jurczyk – 8178-9815
kjurczyk@ufl.edu

B+ Tree Program Structure

Function Prototypes

The following functions exist within my B+ Tree program:

bplustree.cpp

```
int main(int argc, char ** argv)
```

Returns a 0 when the function is done running. Takes the file and funnels the input to the appropriate functions. Receives output from the functions and funnels it into output_file.txt. The filename should be given when calling the program – this is an argument.

methods.h

```
string getCommand(string s)
```

Return as a string the command given in the input file.

```
string getFirstNum(string s)
```

Return as a string the first number after the parentheses if given a string line.

```
string getSecondNum(string s)
```

Return as a string the second number after the parentheses if given a string line.

```
string doStuffWithCommand(string instructions[3], BPTree &bpt)
```

Return as a string whatever the B+ tree outputs, given an array containing the [instruction, first number, second number] in that order, and also a pointer to the instance of the B+ tree being used.

```
void BPTree::deletePair(int key) // deletes a pair from a node and is responsible for
calling all other functions pertaining to delete
void BPTree::mergeWithSibling(nodePointer node, nodePointer parent, nodePointer siblin
g, nodePointer parentOfSibling) // merge a sibling into a node, and make sure that bot
h parents are ok
void BPTree::updateGrandparent(nodePointer node) // update the keys of the grandparent
 to make sure they are alright
void BPTree::percolateUpwards(nodePointer parent) // checks if the parent is legal and
 if not, redoes the merge
void BPTree::insertChildrenIntoIndexNode(nodePointer node, nodePointer receiver) // in
sert children from a given node into the receiving node which has too few children
void BPTree::insertParentKeysIntoIndexNode(nodePointer parent, nodePointer receiver) /
/ pull the parent's keys down into the index node given
BPTree::nodePointer BPTree::getLegalIndexSibling(nodePointer index) // returns a nodeP
ointer to the best sibling to borrow from (built for index nodes because leaf nodes ha
ve sibling pointers)
void BPTree::deleteNode(nodePointer node, nodePointer parent)   // deletes a node from
 its parent (over-
writes that child pointer with another to make the parent forget it existed)
BPTree::nodePointer BPTree::sharedElder(nodePointer a, nodePointer b) // finds the you
ngest shared elder of a and b (e.g. if a and b are cousins, a pointer to the grandpare
nt would be returned)
```

```
void BPTree::insertAsReplacement(nodePointer node, nodePointer parent, nodePointer sibling, bool isBigger)  // replaces keys/values in one node with keys/values with another node
void BPTree::recalculateParentKeys(nodePointer parent) // recalculates the keys in the given node using the array of children
bool BPTree::deleteKeyAndKeyValuePair(nodePointer toBeDeletedFrom, int key)  // toBeDeletedFrom is a leaf - deletes a key and keyValue pair from a leaf node
bool BPTree::deleteKey(nodePointer toBeDeletedFrom, int key) // deletes the key from a node & decreases numKey
bool BPTree::deleteKeyValuePair(nodePointer toBeDeletedFrom, int key, int numKeyValues) // deletes a key and keyValue pair from a leaf node given a specific number of key values
int BPTree::determineChildNum(nodePointer parent, nodePointer child) // determines the number of the child in the parent's children array
```

search.h

```
std::string BPTree::search(int key) // returns the value at the key location
std::string BPTree::search(int key1, int key2) // searches for the range of values which correspond to keys that are >key1 and <key2
```

tree.h

```
void initialize(int m)  // initializes the m-way tree
void insert(int key, float value) // insert a key-value pair into a leaf node
void checkAllSiblings(nodePointer first, nodePointer last)  // checks all the sibling pointers, starting at the smallest node
void assignSiblingsDuringInsert(nodePointer parent) // when you insert a new node, assign the siblings to each other (nextLeaf, prevLeaf)
nodePointer determineLeafNode(int key)  // determine the appropriate node of a given key (used for insert, search, delete)
void stepAfterSplit(nodePointer original, nodePointer large, int numToBeMerged) // now that you have split the node correctly, check if the original node used to be the root (isRoot). if it was not the root, you need to find the parent of the current leaf
void updateParent(nodePointer toBeUpdated)  // takes a given node and splits it into a smaller and larger node, then calls the step after split so we are left with one node
nodePointer initializeNode()  // initializes a node with numKeys and also with the sizes of all the arrays - returns the new node
nodePointer findParent(nodePointer child) // returns the parent of a given node. If the parent could not be found, return nullptr
nodePointer determineWhichChildToTravelDown(nodePointer parent, int keyBeingSearchedFor)  // goes through the given node and compares the keys with the key being searched for. Finds and returns a pointer to the child which should be travelled down next
```

```
void sortChildren(nodePointer *arr, int len)  // go through the array of children and sort them according to their smallest keys
void sortKeyValues(Pair *arr, int len) // go through the keyValue pairs and sort them according to their smallest keys
void sortKeys(int *arr, int len) // go through the array of keys and sort them according to the smallest keys
void printNode(string name, nodePointer node) //  prints a given node with its name
void printNodePairs(string name, nodePointer node) // prints the given node's key-value pairs
void printChildren(string name, nodePointer node) // prints all the children of a given node on a new line
void printTree()  // print the tree starting at the root, for maximum of 3 levels
void printLinkedList(int key1)  // prints the linked list starting with key1
void printLinkedListGivenPointer(nodePointer startHere) // print linked list starting at given nodePointer
void printLinkedListPairs(int key1) // print linked list starting with key1
```

Structure of the Program

The program is based in multiple files. The main function is located in bplustree.cpp.
Main accepts one additional argument, which should be the file name of the input (if not, the program will simply print out a statement telling the user that the file was not opened.

Main interacts with the functions in methods.h to determine which commands need to be run and to run the commands it determines. Methods.h in turn interacts with all other header files. Methods.h only calls the following functions: search(key1), search(key1,key2), insert(key,value), delete(key), and initialize(m).

At the beginning of bplustree.cpp is an insert statement to include methods.h. Methods.h includes search.h. Search.h includes delete.h. Delete.h includes tree.h. Tree.h includes all the other include functions such as iostream, vector, etc.

There are also many lines in the code which start with "if(DEBUG)" which are useful for the user to see what the program does and how the B+ tree looks at specific points in time. At the moment DEBUG is set to false, however, the user can change DEBUG to true at the top of the tree.h file.

tree.h

Class Pair -holds an int key and a float value. Allows editing and printing.
Class BPTree – the heart of the program. Contains (please also see in-line comments for descriptions of the variables and functions listed):

- Struct nodeObject which is used for all nodes (root, index, and leaf)

```
-       int numKeys = 0;  // number of keys in this node
-       int numKids = 0;  // number of children in the node (numKids - 1 = numKeys)
-       nodeObject *prevLeaf = nullptr; // holds a pointer to the previous leaf, if
    this is a leaf node - otherwise we won't use this
-       nodeObject *nextLeaf = nullptr; // holds a pointer to the next leaf, if thi
   s is a leaf node - otherwise we won't use it
-       // we also won't know until later how many keys will be stored in here
-       int *keys;           // should be empty if this is a leaf node
-       Pair *keyValues;     // should be empty if this is an index node
-       nodeObject **children; // an array which holds pointers to the children - s
   et it to null when you can
-       bool isLeaf;  // true if this node is a leaf
-       bool isRoot;  // true if this node is a root
```

- Other variables used for the running of the program

```
-     typedef nodeObject *nodePointer; //define pointers to node objects - makes it
    easier for me later on
-     nodePointer root; // pointer to the root node
-     // least ceil(m/2) children, and at most m children
-     int minKids = 0; // at least ceil(m/2) children
-     int maxKids = 1; // at most m children
-     // there can be between ceil(m/2)-1 keys and m-1 keys
-     int minKeys = 0;  // ceil(m/2)-1 keys
-     int maxKeys = 1;  // m-1 keys
```

- Functions used within the program

```
-     void deletePair(int key); // deletes a pair from a node and is responsible fo
    r calling all other functions pertaining to delete
-     std::string search(int key);  // searches for the value which corresponds to
    key
-     std::string search(int key, int key2);  // searches for the range of values w
    hich correspond to keys that are >key1 and <key2
-     bool deleteKeyAndKeyValuePair(nodePointer toBeDeletedFrom, int key);  // dele
    tes a key and keyValue pair from a leaf node
-     bool deleteKey(nodePointer toBeDeletedFrom, int key); // deletes the key from
     a node & decreases numKey
-     bool deleteKeyValuePair(nodePointer toBeDeletedFrom, int key, int numKeyValue
    s);  // deletes a keyValue from a leaf node
-     int determineChildNum(nodePointer parent, nodePointer child); // determines t
    he number of the child in the parent's children array
-     void insertAsReplacement(nodePointer issue, nodePointer parentOfIssue, nodePo
    inter sibling, bool isBigger); // replaces keys/values in one node with keys/va
    lues with another node
-   void recalculateParentKeys(nodePointer parent); // recalculates the keys in the
     given node using the array of children
-     void deleteNode(nodePointer node, nodePointer parent);  // deletes a node fro
    m its parent (over-
    writes that child pointer with another to make the parent forget it existed)
-     nodePointer getLegalIndexSibling(nodePointer index);    // returns a nodePoin
    ter to the best sibling to borrow from (built for index nodes because leaf node
    s have sibling pointers)
-     void insertParentKeysIntoIndexNode(nodePointer parent, nodePointer receivingN
    ode);  // pull the parent's keys down into the index node given
-     void insertChildrenIntoIndexNode(nodePointer node, nodePointer receiver);
          // insert children from a given node into the receiving node which has t
    oo few children
-     void percolateUpwards(nodePointer parent);  // checks if the parent is legal
    and if not, redoes the merge
-     nodePointer sharedElder(nodePointer a, nodePointer b);  // finds the youngest
     shared elder of a and b (e.g. if a and b are cousins, a pointer to the grandpa
    rent would be returned)
-     void mergeWithSibling(nodePointer node, nodePointer parent, nodePointer sibli
    ng, nodePointer parentOfSibling);  // merge a sibling into a node, and make sur
    e that both parents are ok
-     void updateGrandparent(nodePointer node); // update the keys of the grandpare
    nt to make sure they are alright
-   void initialize(int m)  // initializes the m-way tree
-   void insert(int key, float value) // insert a key-value pair into a leaf node
-   void checkAllSiblings(nodePointer first, nodePointer last)  // checks all the s
    ibling pointers, starting at the smallest node
-   void assignSiblingsDuringInsert(nodePointer parent) // when you insert a new no
    de, assign the siblings to each other (nextLeaf, prevLeaf)
-   nodePointer determineLeafNode(int key)  // determine the appropriate node of a
    given key (used for insert, search, delete)
-   void stepAfterSplit(nodePointer original, nodePointer large, int numToBeMerged)
     // now that you have split the node correctly, check if the original node used
     to be the root (isRoot). if it was not the root, you need to find the parent o
    f the current leaf
-   void updateParent(nodePointer toBeUpdated)  // takes a given node and splits it
     into a smaller and larger node, then calls the step after split so we are left
     with one node
-   nodePointer initializeNode()  // initializes a node with numKeys and also with
    the sizes of all the arrays - returns the new node
-   nodePointer findParent(nodePointer child) // returns the parent of a given node
    . If the parent could not be found, return nullptr
-   nodePointer determineWhichChildToTravelDown(nodePointer parent, int keyBeingSea
    rchedFor)  // goes through the given node and compares the keys with the key be
    ing searched for. Finds and returns a pointer to the child which should be trav
    elled down next
-   void sortChildren(nodePointer *arr, int len)  // go through the array of childr
    en and sort them according to their smallest keys
-   void sortKeyValues(Pair *arr, int len) // go through the keyValue pairs and sor
    t them according to their smallest keys
-   void sortKeys(int *arr, int len) // go through the array of keys and sort them
    according to the smallest keys
-   void printNode(string name, nodePointer node) //  prints a given node with its
    name
-   void printNodePairs(string name, nodePointer node) // prints the given node's k
    ey-value pairs
-   void printChildren(string name, nodePointer node) // prints all the children of
     a given node on a new line
-   void printTree()  // print the tree starting at the root, for maximum of 3 leve
    ls
-   void printLinkedList(int key1)  // prints the linked list starting with key1
-   void printLinkedListGivenPointer(nodePointer startHere) // print linked list st
    arting at given nodePointer
-   void printLinkedListPairs(int key1) // print linked list starting with key1
```

delete.h

All the functions in the following header files were already defined in the tree.h file and listed in the “Function Prototypes” section. However, just to be clear, the following full functions can be found in the delete.h file (with in-line descriptions of the functions):

```
void BPTree::deletePair(int key) // deletes a pair from a node and is responsible for
calling all other functions pertaining to delete
```

```
void BPTree::mergeWithSibling(nodePointer node, nodePointer parent, nodePointer sibling, nodePointer parentOfSibling) // merge a sibling into a node, and make sure that both parents are ok
void BPTree::updateGrandparent(nodePointer node) // update the keys of the grandparent to make sure they are alright
void BPTree::percolateUpwards(nodePointer parent) // checks if the parent is legal and if not, redoes the merge
void BPTree::insertChildrenIntoIndexNode(nodePointer node, nodePointer receiver) // insert children from a given node into the receiving node which has too few children
void BPTree::insertParentKeysIntoIndexNode(nodePointer parent, nodePointer receiver) // pull the parent's keys down into the index node given
BPTree::nodePointer BPTree::getLegalIndexSibling(nodePointer index) // returns a nodePointer to the best sibling to borrow from (built for index nodes because leaf nodes have sibling pointers)
void BPTree::deleteNode(nodePointer node, nodePointer parent)   // deletes a node from its parent (over-writes that child pointer with another to make the parent forget it existed)
BPTree::nodePointer BPTree::sharedElder(nodePointer a, nodePointer b) // finds the youngest shared elder of a and b (e.g. if a and b are cousins, a pointer to the grandparent would be returned)
void BPTree::insertAsReplacement(nodePointer node, nodePointer parent, nodePointer sibling, bool isBigger)  // replaces keys/values in one node with keys/values with another node
void BPTree::recalculateParentKeys(nodePointer parent) // recalculates the keys in the given node using the array of children
bool BPTree::deleteKeyAndKeyValuePair(nodePointer toBeDeletedFrom, int key)  // toBeDeletedFrom is a leaf - deletes a key and keyValue pair from a leaf node
bool BPTree::deleteKey(nodePointer toBeDeletedFrom, int key) // deletes the key from a node & decreases numKey
bool BPTree::deleteKeyValuePair(nodePointer toBeDeletedFrom, int key, int numKeyValues) // deletes a key and keyValue pair from a leaf node given a specific number of key values
int BPTree::determineChildNum(nodePointer parent, nodePointer child) // determines the number of the child in the parent's children array
```

search.h

```
std::string BPTree::search(int key) // returns the value at the key location
std::string BPTree::search(int key1, int key2) // searches for the range of values which correspond to keys that are >key1 and <key2
```