Trabajo Practico Árboles B y B+
Reynaldo Cusi

Arbol B

```
//                [] []
//          []       []     []
//      []        []      []     []   []
// []      [] []      []     []    []    []

// h = O(log_m(N))
#include <iostream>
#include <vector>
#define ORDER 3

using namespace std;

enum state_t
{
  OVERF,
  UNDERF,
  OK
};

class btree
{
  struct node
  {
    std::vector<char> values;
    std::vector<node *> children;
    int count{0};
    node() : values(ORDER + 1), children(ORDER + 2, nullptr)
    {
    }

    void insert_into(int index, int value)
    {
      // [10, 20, 30, 40] ->
      // [10, 15,  20, 30, 40]
      // value = 15, index = 1

      int i = count;
```

```
    for (; i > index; i--)
    {
      values[i] = values[i - 1];
      children[i + 1] = children[i];
    }
    values[i] = value;
    children[i + 1] = children[i];
    this->count++;
  }
  state_t insert(int value)
  {
    // base
    int children_index = 0;
    while (value > values[children_index] && children_index < count)
    {
      children_index++;
    }
    if (children[children_index] == nullptr)
    { // leaf node
      insert_into(children_index, value);
    }
    else
    { // index node
      auto state = children[children_index]->insert(value);
      if (state == state_t::OVERF)
      {
        split(this, children_index);
      }
    }
    return count > ORDER ? state_t::OVERF : state_t::OK;
  }

  void split(node *ptr, int index)
  {
    node *node_in_overflow = ptr->children[index];
    node *node1 = new node();
    node *node2 = new node();
    int n = node_in_overflow->count;
    int i;
    for (i = 0; i < n / 2; i++)
    {
      node1->children[i] = node_in_overflow->children[i];
      node1->values[i] = node_in_overflow->values[i];
```

```
        node1->count++;
      }
      node1->children[i] = node_in_overflow->children[i];
      ptr->insert_into(index, node_in_overflow->values[i]);
      i++; // skip middle
      int j = 0;
      for (; i < n; i++, j++)
      {
        node2->children[j] = node_in_overflow->children[i];
        node2->values[j] = node_in_overflow->values[i];
        node2->count++;
      }
      node2->children[j] = node_in_overflow->children[i];
      ptr->children[index] = node1;
      ptr->children[index + 1] = node2;
    }
  };

public:
  void insert(int value)
  {
    auto state = root.insert(value);
    if (state == state_t::OVERF)
    {
      splitRoot();
    }
  }
  void splitRoot()
  {
    std::cout << "===============\n";

    node *node_in_overflow = &root;
    int index = 0;
    node *ptr = new node();
    node *node1 = new node();
    node *node2 = new node();
    int n = node_in_overflow->count;
    int i;
    for (i = 0; i < n / 2; i++)
    {
      node1->children[i] = node_in_overflow->children[i];
      node1->values[i] = node_in_overflow->values[i];
      node1->count++;
```

```
    }
    node1->children[i] = node_in_overflow->children[i];
    ptr->values[index] = node_in_overflow->values[i];

    i++;
    int j = 0;
    for (; i < n; i++, j++)
    {
      node2->children[j] = node_in_overflow->children[i];
      node2->values[j] = node_in_overflow->values[i];
      node2->count++;
    }
    node2->children[j] = node_in_overflow->children[i];
    ptr->children[index] = node1;
    ptr->children[index + 1] = node2;
    ptr->count = 1;
    root = *ptr;
  }

  bool find(char &value)
  {
    return find(&root, value);
  }

  bool find(node *ptr, char &value)
  {
    int pos = 0;
    while (pos < ptr->count && ptr->values[pos] < value)
      ++pos;

    if (pos < ptr->count && ptr->values[pos] == value)
      return true;

    return ptr->children[pos] ? find(ptr->children[pos], value) :
false;
  }

  void print()
  {
    std::cout << "===============\n";
    print(&root, 0);
    std::cout << "\n";
  }
```

```
  void print(node *ptr, int level)
  {
    if (ptr)
    {
      int i;
      for (i = ptr->count - 1; i >= 0; i--)
      {
        print(ptr->children[i + 1], level + 1);
        for (int k = 0; k < level; k++)
          std::cout << "    ";
        std::cout << ptr->values[i] << "\n";
      }
      print(ptr->children[i + 1], level + 1);
    }
  }

private:
  node root;
};
#include <string>

int main()
{
  btree bt;
  std::string data = "abcdefg";
  for (auto &&v : data)
  {
    bt.insert(v);
  }
  bt.print();
  char a = 'd';
  cout << bt.find(a) << endl;
  char b = 'z';
  cout << bt.find(b) << endl;
  return 0;
}
```

Arbol B+

```
//                [] []
//          []       []     []
//      []        []      []     []   []
// []      [] []      []     []    []    []

// h = O(log_m(N))
#include <iostream>
#include <vector>
#define ORDER 3

using namespace std;

enum state_t
{
  OVERF,
  UNDERF,
  OK
};

class btree
{
  struct node
  {
    std::vector<char> values;
    std::vector<node *> children;
    int count{0};
    node() : values(ORDER + 1), children(ORDER + 2, nullptr)
    {
    }
    void insert_into(int index, int value)
    {
      // [10, 20, 30, 40] ->
      // [10, 15,  20, 30, 40]
      // value = 15, index = 1

      int i = count;
      for (; i > index; i--)
      {
        values[i] = values[i - 1];
        children[i + 1] = children[i];
      }
```

```
    values[i] = value;
    children[i + 1] = children[i];
    this->count++;
  }
  state_t insert(int value)
  {
    // base
    int children_index = 0;
    while (value > values[children_index] && children_index < count)
    {
      children_index++;
    }
    if (children[children_index] == nullptr)
    { // leaf node
      insert_into(children_index, value);
    }
    else
    { // index node
      auto state = children[children_index]->insert(value);
      if (state == state_t::OVERF)
      {
        split(this, children_index);
      }
    }
    return count > ORDER ? state_t::OVERF : state_t::OK;
  }

  void split(node *ptr, int index)
  {
    node *node_in_overflow = ptr->children[index];
    node *node1 = new node();
    node *node2 = new node();
    int n = node_in_overflow->count;

    int i;
    for (i = 0; i < n / 2; i++)
    {
      node1->children[i] = node_in_overflow->children[i];
      node1->values[i] = node_in_overflow->values[i];
      node1->count++;
    }
    node1->children[i] = node_in_overflow->children[i];
    ptr->insert_into(index, node_in_overflow->values[i]);
```

```
      if (node_in_overflow->children[0] != nullptr)
        i++;
      int j = 0;
      for (; i < n; i++, j++)
      {
        node2->children[j] = node_in_overflow->children[i];
        node2->values[j] = node_in_overflow->values[i];
        node2->count++;
      }
      node2->children[j] = node_in_overflow->children[i];
      ptr->children[index] = node1;
      ptr->children[index + 1] = node2;
    }
  };

public:
  void insert(int value)
  {
    auto state = root.insert(value);
    if (state == state_t::OVERF)
    {
      splitRoot();
    }
  }
  void splitRoot()
  {
    std::cout << "===============\n";

    node *node_in_overflow = &root;
    int index = 0;
    node *ptr = new node();
    node *node1 = new node();
    node *node2 = new node();
    int n = node_in_overflow->count;
    int i;
    for (i = 0; i < n / 2; i++)
    {
      node1->children[i] = node_in_overflow->children[i];
      node1->values[i] = node_in_overflow->values[i];
      node1->count++;
    }
    node1->children[i] = node_in_overflow->children[i];
    ptr->values[index] = node_in_overflow->values[i];
```

```
    if (node_in_overflow->children[0] != nullptr)
      i++;
    /* i++; */
    int j = 0;
    for (; i < n; i++, j++)
    {
      node2->children[j] = node_in_overflow->children[i];
      node2->values[j] = node_in_overflow->values[i];
      node2->count++;
    }
    node2->children[j] = node_in_overflow->children[i];
    ptr->children[index] = node1;
    ptr->children[index + 1] = node2;
    ptr->count = 1;
    root = *ptr;
  }

  bool find(char &value)
  {
    return find(&root, value);
  }

  bool find(node *ptr, char &value)
  {
    int pos = 0;
    while (pos < ptr->count && ptr->values[pos] < value)
      ++pos;

    if (pos == ptr->count)
      return ptr->children[pos] ? find(ptr->children[pos], value) :
false;

    if (ptr->values[pos] == value)
      return true;

    if (ptr->children[pos] == NULL)
      return false;
    else
      find(ptr->children[pos], value);
  }

  vector<char> find_range(char min, char max)
```

```
{
  vector<char> content;
  find_range(&root, min, max, content);
  return content;
}

void find_range(node *ptr, int min, int max, vector<char> &content)
{
  if (ptr->children[0] == nullptr)
  {
    for (int i = 0; i < ptr->count; i++)
      if (ptr->values[i] >= min && ptr->values[i] <= max)
        content.push_back(ptr->values[i]);
    return;
  }
  int pos_i = 0;
  int pos_j = 0;

  while (pos_i < ptr->count && ptr->values[pos_i] <= min)
  {
    pos_i++;
    pos_j++;
  }

  while (pos_j < ptr->count && ptr->values[pos_j] <= max)
    pos_j++;

  for (int i = pos_i; i <= pos_j; i++)
    find_range(ptr->children[i], min, max, content);
}

void print()
{
  std::cout << "===============\n";
  print(&root, 0);
  std::cout << "\n";
}
void print(node *ptr, int level)
{
  if (ptr)
  {
    int i;
    for (i = ptr->count - 1; i >= 0; i--)
```

```
      {
        print(ptr->children[i + 1], level + 1);
        for (int k = 0; k < level; k++)
          std::cout << "    ";
        std::cout << ptr->values[i] << "\n";
      }
      print(ptr->children[i + 1], level + 1);
    }
  }

private:
  node root;
};
#include <string>

int main()
{
  btree bt;
  std::string data = "ABCDEFGH";
  for (auto &&v : data)
  {
    bt.insert(v);
  }
  bt.print();
  for (auto &&i : bt.find_range('B', 'D'))
  {
    cout << i << " ";
  }
  cout << endl;
  return 0;
}
```