CES12 Lab Balanced Tree

VERSÃO 200416

Nada mudou além de comentários em IndexPointsAluno.cpp para explicar melhor a semântica do método find em relação ao intervalo ser aberto ou fechado.

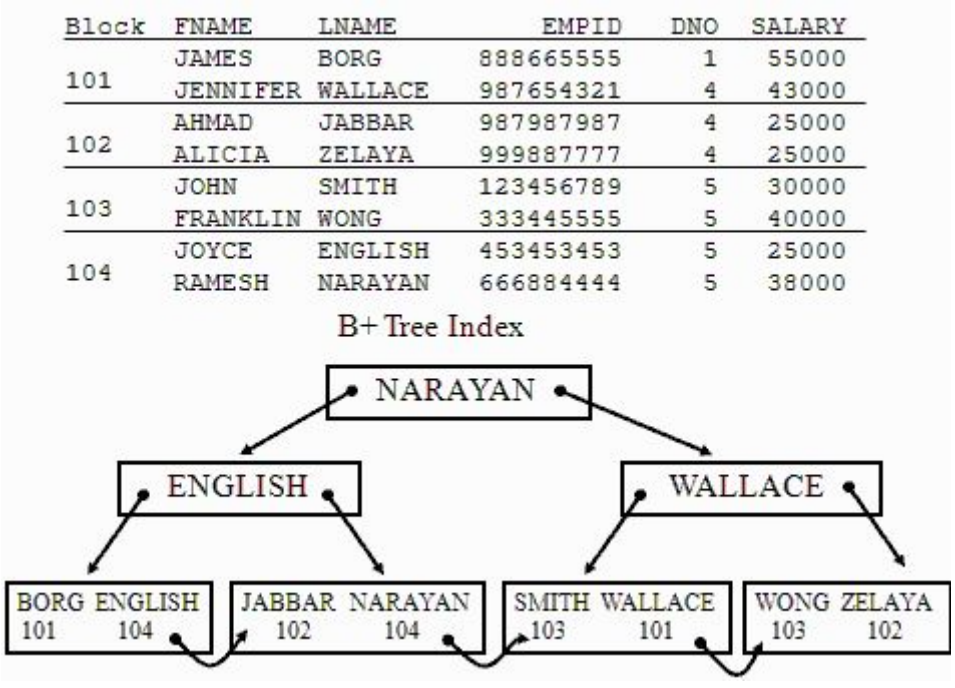
Objetivo

Implementar uma árvore balanceada e verificar que as operações de busca e inserção são O(log n)

Conceitos

Índice

Suponha que os dados estão em um vetor ou em um grande arquivo em disco que não cabe na memória, e desejamos realizar buscas por quaisquer campos - no nosso caso há 3 coordenadas, mais uma categoria.
Ou ainda, podemos buscar por algum outro critério - por exemplo, “busque todos os pontos a menos que 10 m do ponto (10,20,1)” - precisamos calcular a distância para cada ponto (O(n)), mas depois disso, armazenando-se estas distâncias, a busca deve ser similar à qualquer busca por um campo direto.



Block	FNAME	LNAME	EMPID	DNO	SALARY
101	JAMES	BORG	888665555	1	55000
	JENNIFER	WALLACE	987654321	4	43000
102	AHMAD	JABBAR	987987987	4	25000
	ALICIA	ZELAYA	999887777	4	25000
103	JOHN	SMITH	123456789	5	30000
	FRANKLIN	WONG	333445555	5	40000
104	JOYCE	ENGLISH	453453453	5	25000
	RAMESH	NARAYAN	666884444	5	38000

Uma busca rápida O(log n) pode ser implementada por várias estruturas de dados. A figura acima, por exemplo, mostra uma B+Tree que permite encontrar empregados pelo

sobrenome. Mas a estrutura exata não é importante - o ponto do conceito de índice é: não precisamos manter uma cópia completa dos dados em uma estrutura para ordenar/buscar dados pelo campo x, e outra cópia completa dos dados para fazer buscas pelo campo y, etc. Isso seria um desperdício de espaço!

Cada estrutura de dados necessária para realizar buscas por um determinado critério precisa apenas armazenar, para cada elemento, um índice que permita encontrar este elemento na estrutura de dados principal (ou arquivo). Note na figura como a árvore apenas contêm chaves (strings) e índices (números inteiros), onde os índices representam os números da coluna “block”, que podem ser usados para encontrar os dados completos.

Este índice poderia ser um ponteiro para o dado, a posição do dado em um arquivo grande, o número do disco/setor em um sistema de arquivos, etc; Qualquer informação que permita encontrar o registro completo do elemento desejado rapidamente. No nosso caso, onde os dados estão em um vetor na memória, simplesmente um inteiro (long) que é a posição do dado no vetor.

Árvore balanceada

Uma árvore é balanceada se as diferenças entre as alturas de quaisquer sub-árvores são limitadas por algum critério que permita garantir que as buscas continuam O(log n) no pior caso. Obviamente, como inserções e remoções podem desbalancear a árvore, são necessários algoritmos para detectar o desbalanceamento e re-balancear a árvore - sem comprometer a complexidade das operações.
Há vários algoritmos com vários critérios mais ou menos rigorosos - árvores AVL, rubro-negras, B e variações como B+, B*, etc.

Buscar valores em um intervalo em uma árvore

Temos uma árvore, e, ao invés de buscar por uma chave específica, buscamos por todos os valores de chave em um intervalo [first, last]. Uma primeira implementação seria um percurso in-order que imprime ou retorna apenas os valores que pertencem ao intervalo. Isso seria O(n), o que é muito ruim se há muitas chaves armazenadas mas o número de chaves a serem retornadas é pequeno - o que é o nosso caso e o caso mais comum.

Atenção ao seguinte exercício de CES11[1], que é exatamente busca de intervalos:

Weiss 4.32 (livro 2d edição)
Escreva uma função que tome como entrada uma árvore binária de busca T, e duas chaves k1 e k2, ordenadas de forma que k1 ≤ k2, e imprima todos os elementos X em T tal que k1 ≤ chave(X) ≤ k2. Não assuma nenhuma informação sobre o tipo de chave exceto que chaves podem ser comparadas e ordenadas.
O programa deverá executar em O(K+logN), onde K é o número de chaves impressas.

[1] Um dos pontos deste lab é mostrar que exercícios de CES11 podem parecer esdrúxulos para os bixos, mas muitos tem aplicação prática.

A solução convencional é implementar uma variação do percurso in-order, onde o percurso não visita um filho se há certeza que ***todos*** os elementos nessa sub-árvore são menores que k1 (first), OU se todos os elementos nessa subárvore são maiores que k2 (last). Pense em qual a condição necessária - depois disso a implementação é fácil.
Nesse caso, “descer” da raiz até k1 (first) demora O(log n) e continuar o in-order até k2 (last) demora O(k). Note que o inorder não apenas ignora ramos da árvore com valores menores que k1, também ignora ramos da árvore com valores maiores que k2.

Note que $O(K + \log n) \notin \Theta(K \log n)$, o que elimina algumas possibilidades de implementar uma função que encontra o sucessor de um nó da árvore.

Pode-se também resolver o problema de outra forma: durante a construção da árvore, fazer com que cada nó tenha ponteiros para os seus antecessores e sucessores, o que não é tão fácil para árvores Red-Black e AVL devido às sub-árvores e rotações da árvore balanceada. Mas, é exatamente o objetivo da B+tree (olhe a figura exemplo acima).

Implementação

No diretório src, há a classe IndexPointsAluno. Ela contém as seguintes operações:

```
long size()
void add (float key, long idx )
void find(std::vector<long> &res, float first, float last ) ;
```

size() simplesmente retorna o número total de elementos no índice. (não precisa implementar uma busca, pode apenas implementar um contador)

add() recebe uma chave e um valor associado a serem inseridos no índice.

Chave é o campo a ser usado para indexar. Em uma busca, a classe recebe valores desejados de chave. Idx é o valor associado à chave, que deve ser retornado como resultado da busca - que corresponde ao índice no vetor onde os pontos completos estão armazenados.

find() recebe como entrada um valor mínimo (first) e um valor máximo (last), e preenche o vetor res com todos os elementos cuja chave está entre first e last. Note que a chave é float, então comparação com == não faz sentido. Quando a função find() terminar, o vetor res deve ter o tamanho correspondente ao número de elementos encontrados.

Note que quem define o critério para gerar as chaves (e.g., se as chaves são valores do campo x) é o usuário da classe, ou seja, o código que chama esta função. Esta classe apenas mapeia um intervalo de float em um conjunto de valores long

add() deve executar em O(log n)
find() deve executar em O(k + log n) conforme discutido acima.

Note que não há operação de remoção, e não é necessário implementar[2].

Vale implementar qualquer árvore balanceada (e.g. B, AVL), mas eu sugiro árvores vermelho-e-preto[3].

Vale: se basear em código ou pseudo-código em livros (inclusive sobre a estrutura de dados em questão, e inclusive trocar ou recomendar bibliografia)
Não vale: copy-paste de código pronto da net ou outrem que implementem a estrutura de dados em questão.

Arquivo de entrada

O arquivo de entrada para VoidSphereSelection está armazenado no diretorio data e compactado no formato bz2, utilize bunzip2 para descompactar. Se não descompactar, ha segfault no teste OakReadFile, por não encontrar o aquivo de entrada..
Porque bunzip2 e não zip? Porque comprime mais e queria economizar espaço!

Testes

A classe IndexPointsAluno foi entregue funcionando baseada na classe std::multimap da STL, que usualmente implementa uma árvore vermelho-e-preto.
O que o aluno deve fazer é retirar a classe std::multimap e implementar funcionalidade equivalente.

Uma outra classe, no diretório lib, chamada IndexPointsStlMultimap, é uma implementação igual que não deve ser modificada pois é usada para comparação. Ambas são filhas da classe abstrata IndexPointsAbstract.

O código dos testes realiza as mesmas operações com IndexPointsStlMultimap e IndexPointsAluno, ou seja, com uma implementação baseada na stl e a sua implementação. Ambas devem sempre apresentar os mesmos resultados.

OakByNorm

1. Divide os dados em 100 partes
2. Insere 1/100 dos pontos, usando a norma dos pontos como chave, na implementação de referência e na do aluno.
3. Sorteia um ponto aleatório
4. Busca pelo ponto sorteado, na implementação de referência e na do aluno
5. Escreve em arquivo oak_searck.csv uma linha com o formato:
    a. <passo de 1 a 100 >, <# elementos inseridos>, <tempo busca STL>, <tempo busca aluno>, <tempo inserções STL>, <tempo inserções aluno>

[2] E remoção é - de longe - a mais difícil de implementar. No meu tempo de bixo em CES11, eu implementei B-tree com remoção.
[3] Se algum torcedor do fluminense for ideologicamente contra, escolha outra árvore.

6. Volta ao passo 2

Portanto, no arquivo teremos o tempo para buscas para 100 tamanhos diferentes de vetor, até 1M elementos.

A sua implementação deve apresentar curvas de tempo semelhantes à implementação da STL, ou seja, O(log n).

“Busca por um ponto’ na verdade significa busca por um pequeno intervalo em torno do ponto procurado. A função para busca com tolerância já está implementada na classe abstrata-pai. O intervalo é pequeno o suficiente para a resposta conter muito poucos pontos.

VoidSphereSelection

1. Insere todos os pontos (1M) em ambas as implementações, usando a distância em relação a um ponto pré-definido como chave
2. Seleciona, em ambas as implementações, pontos entre 5 a 13 metros do ponto central.
3. Escreve 2 arquivos contendo os pontos selecionados, com linhas no formato
    a. <coordenada X> <coord Y> <coord Z>
4. Imprime o tempo gasto na seleção.

A sua implementação deve ser O(k) e não O(n).
A sua implementação deve retornar exatamente o mesmo número de pontos.

O arquivo de saída pode ser verificado em plotadores de nuvens de pontos (e.g., meshlab).

O ponto e as distâncias foram escolhidos de forma a aparecer 2 carros e uma árvore em uma rua, que devem ser visíveis no plot3D. Verificarei com meshlab na saída de cada um.

Testes que não valem nota

IndexFileALU

Carrega um arquivo menor (100 pontos), insere os 100 pontos na estrutura do aluno e realiza uma busca. Verifica se o ponto buscado foi encontrado.

IndexFile

Similar a IndexfileALU mas usando a implementação de referência.

readAscFile

Testa a leitura de um pequeno arquivo. Um erro aqui pode indicar problemas de permissão, ou que o arquivo de entrada não foi descompactado.

Relatório

1) Descreva a estrutura que escolheu:

- Qual estrutura (e.g. árvore vermelho-preto)
- (se houver) Qual livro utilizou como base da implementação
    - O pseudo-código que mostrei na aula foi retirado do Corben - que ainda contém o pseudo-código das rotações e inserção, mostra devagar os conceitos básicos como o papel dos nós red-black e dos nodes NIL, e prova a corretude do algoritmo usando as invariâncias do loop.
    - Talvez o próprio Sedgewick tenha escrito uma boa referência também, mas o livro que tenho dele não tem nada. Podem haver outros livros dele.
- Alguma consideração que ajude a entender o seu código (implementou algo diferente do usual? Há classes diferentes que mereçam um diagrama de classes ou uma explicação? Vale desenhos a mão fotografados ou diagramas feitos em software.).

2) Mostre na forma de gráficos as curvas de tempo do teste OakByNorm, comparando a sua implementação com a minha implementação baseada em stl::multimap.
Mostre que as buscas **e** inserções são O(log n) em ambas as implementações - use um programa de fit em função logaritmica.

Nota

5 OakbyNorm O(log n)
4 Voidsphere O(k + log n)
1 Apresentação e legibilidade do relatório.

Dicas:

Plot

O arquivo .asc (3 colunas com coordenadas XYZ, um ponto por linha) pode ser plotado como nuvem de pontos no matlab, com o comando scatter3. É possível também usar plot3, mas este é mais pesado. Cada ponto no plot3 possui mais propriedades, e isso aumenta o uso de memória. O scatter3 é mais indicado para nuvem de pontos onde cada ponto é basicamente apenas uma coordenada XYZ e uma cor.

Tempo

É perfeitamente possível (e aconteceu várias vezes) que a sua implementação seja mais rápida do que a implementação MMP (usando std::multimap). Isto não é surpresa pois a STL é mais genérica e as operações gastam tempo devido aos templates, uma maior hierarquia de classes, etc.

Errata Cormen em Portugues

Erro na 3a edição em português do Cormen. A 3a edição em inglês está correta. Função RB-insert, linha 11, onde está

```
elseif z.chave < z.chave
```
deve ser
```
elseif z.chave < y.chave
```

FAQ

- A função find recebe um vetor de long chamado res. Contudo, no arquivo .cpp, a função find é void e tem uma parte para imprimir os elementos encontrados. Então, é para imprimir os valores encontrados e guardá-los no vetor res?

É só para preencher os o vetor res com os valores encontrados, não para imprimir.

O vetor tem que existir antes de chamar find(), não dá pra chamar esse cabeçalho com ponteiro nulo. Mas ele pode estar vazio (tamanho zero) ou ter um tamanho completamente diferente.

vc pode fazer res.resize() para garantir que ele terá o tamanho correto (se quem chamou já criou o vetor com o tamanho correto, nada acontece), ou chamar res.clear() no início e adicionar ponto por ponto. Alguns testes chamam find mais de uma vez com o mesmo vetor, e checam o tamanho depois. Não é pra acumular resultados de buscas anteriores, é pra terminar a função com o vetor res de tamanho correto.

PS: o nome ‘res’ vem de ‘resposta’ ou ‘resultado’ da busca

- Os elementos de cada nó da árvore (no caso se for rubro-negra) podem ser uma estrutura contendo um float chave, um long que é o elemento, um identificador de cor e ponteiros para o pai, filho esquerdo e direito? Então, pode haver nós diferentes com a mesma chave, certo?

A estrutura da árvore pode aceitar valores repetidos da mesma forma que uma árvore de busca usual. Outras formas de tratar repetição também podem ser implementadas, como fazer cada nó guardar uma lista de elementos.
Outra pergunta é se os testes podem gerar valores repetidos - depende de como geramos as chaves, e, no nosso caso, como podem haver vários pontos com a mesma coordenada

x, ou que por sorte tenham a mesma distância do ponto central, então sim, pode haver repetição.
Aliás, foi por isso que usei stl::multimap ao invés de stl::map, a diferença é justamente que multimap permite repetições.

não ficou clara na apresentação, eu insiro os nós onde especificamente? Pela explicação deu a entender que pode ser em qualquer nó vazio e depois vai se verificando se pode trocar as cores ou rotacionar. É isso mesmo?

Se não ficou claro na discussão inicial sobre árvores de busca em geral, tanto RB-trees quanto B-trees ou AVL-trees e suas variantes, são árvores de busca - portanto a inserção é igual à inserção de árvore de busca, o algoritmo apenas corrige a violação das propriedades que garantem o balanceamento, que pode surgir como resultado da inserção.

a função size, porque ela retorna um valor do tipo long se é para retornar a quantidade de elementos na árvore, ou seja, um int?

É verdade, se o int tem 4 bytes, ja seria suficiente mesmo com 1M pontos. Mas o standard não obriga o int a ser 4 bytes, pode ser 2, o que era comum quando aprendi a programar e isso cria alguma paranoia… De qq forma o long é obrigado a ser 32 bits, e se o objetivo fosse deixar o código super-portável, precisaria deixar long.
https://en.cppreference.com/w/cpp/language/types
Suponho que compiladores para microcontroladores (CPUs menores ou embarcadas) ainda podem definir int como 2 bytes. Mas no nosso caso não precisamos desse nível de paranoia, poderia ser int.

Precisa criar um destrutor para a árvore?

O destrutor é um bom auto teste para vocês. Manter o destrutor funcionando em todas as execuções, sem encontrar nenhum filho null (ponteiro null - é pra apontar para o nó NIL), sem loop infinito (ponteiro para ancestral com deleção bottom-up), sem outros segfaults (ponteiro com valor errado ou 2 ponteiros apontando para o mesmo nó), seria uma validação adicional da árvore e ajuda a encontrar logo bugs na construção ou nos algoritmos de rebalanceamento da árvore. E além disso é uma recursão super fácil de implementar, apenas cuidado em não deletar o nó NIL especial - mantenha um ponteiro separado pra ele e delete ele depois.
Se forem espertos, irão implementar o destrutor antes de implementar o algoritmo de rotação, e manter o destrutor funcionando como teste. Se forem mais espertos ainda e usarem valgrind (muito fácil de aprender o uso básico, mesmo que seja uma ferramenta complexa), ainda pode-se detectar alocações não desalocadas depois do destrutor, que podem também indicar erros na construção na árvore ou nas rotações.
Não testarei se está destruindo corretamente - mas não pode terminar com segfault - o testador tem que terminar a execução.

Precisa manter a ordem dos pontos no resultado da busca?

Ao invés de fazer um inorder esperto, é possível fazer (e alunos já fizeram) um pre-order esperto, também O(k + logn). Como o pre-order coloca na resposta os pais antes dos filhos,

a resposta não saiu em ordem crescente. A especificação poderia exigir que a ordem fosse mantida - mas não exige, então tudo bem. Claro que o desenho da nuvem de pontos é o mesmo independentemente da ordem dos pontos.

O que é o código em IndexPointsStlMultimap::find que chama a stl::multimap ? lower_bound, aquela variavel ‘it’

É um design pattern (DP, ou padrão de projeto) chamado Iterator. Primeiro, várias classes da STL implementam este DP, então poderia trocar o stl::multimap por outra estrutura de dados, sem ter que mudar nada nesse código do IndexPointsStlMultimap::find. Segundo, naturalmente conseguimos acessar apenas os pontos do intervalo em O(K + logn). Terceiro, mesmo se usássemos estruturas de dados indexadas como vetores, não há preocupação ou dificuldade em acertar os valores de início e fim das condições do if - isso elimina bugs.

Pode procurar como implementar, existem sites e livros com detalhes de DPs, seria um bom exercício para aprender OOP e detalhes de C++ ; mas **não é necessário**. Já houveram alunos que conseguiram manter a mesma interface, apenas trocar stl::multimap pela classe deles sem precisar mudar mais nada em IndexPointsAluno.cpp. E outros que implementaram versões mais simplificadas do conceito de iterator.

2019 Bloopers

Não repita (copy-paste) várias linhas de código para criar nós nulos.

Em vários códigos apareceram, várias vezes repetidas, linhas como

```
newnode->campo1 = 0;
newnode->campo2 = 0;
newnode->campo3 = 0;
```

etc.

primeiro, mesmo struct em C++ admite um construtor (objetos são structs com encapsulamento, herança, polimorfismo, etc). Crie um construtor default para criar o nó vazio. Podem haver até vários construtores com argumentos diferentes.
e mesmo que fosse C, seria fácil criar uma função struct Node* criaNovoNo() que deixaria o seu código mais limpo e com menos bugs.
cada copy-paste é uma nova possibilidade de erro de digitação! e código longo é mais difícil de visualizar na tela, abstrair e entender. se ajudem e facilitem a vida de vocês.
hum, e também é possível fazer:

```
a = b = c = d = 0;
```

quando se quer inicializar várias variáveis com o mesmo valor, pois o operador de atribuição também retorna um valor