INF 610 – Estrutura de dados e algoritmos - 2018/2

Lista de exercícios 3

Marcelo de Matos Menezes - 75254

1 – Implementações

As implementações foram feitas em C++ 11 utilizando *templates* para que os multiconjuntos pudessem conter qualquer tipo comparável. Nos arquivos "odamultiset.hpp" e "rbtmultiset.hpp" estão as implementações descritas nas seções 1.1 e 1.2, respectivamente. O arquivo "benchmark.cpp" contém os testes de desempenho cujos resultados são detalhados na seção 3 e o arquivo "debug.cpp" contém um programa para depuração visual de algumas operações implementadas. O *makefile* contém duas diretivas para compilação dos programas utilizando o *GCC*. O código pode ser acessado em [1].

1.1 - *Ordered Dynamic Array*

A primeira abordagem para implementação do multiconjunto foi utilizar um *array* dinâmico ordenado. A ideia é alocar os elementos de forma contínua na memória e expandir a capacidade do *array* quando todas as posições forem ocupadas. Os itens são mantidos ordenados para uma maior eficiência de operações que requerem buscas no conjunto. Além disso, cada elemento do *array* possui um contador para indicar sua quantidade, otimizando memória e tempo para armazenar elementos repetidos. Chamaremos essa implementação de ODA.

1.2 - *Red-Black Tree*

A árvore vermelho-preto é uma árvore binária de pesquisa que utiliza a coloração de seus nós e algumas propriedades para garantir que o caminho da raiz até determinada folha não seja mais que duas vezes o tamanho do caminho da raiz até outra folha qualquer [2]. Isso faz com que a árvore esteja aproximadamente balanceada, garantindo um pior caso da ordem de O(log n) para as operações sobre o multiconjunto. Assim como no *array*, cada nó da árvore implementada possui um contador para otimizar a manipulação de elementos repetidos. Chamaremos essa implementação de RBT.

2 - Análise teórica

Análise das complexidades, considerando *n* como o número de elementos distintos do multiconjunto e comparação como operação básica de todas as funções.

2.1 - Operações do ODA

2.1.1 - Insere

A inserção foi implementada de modo que, ao adicionar um novo elemento, o *array* continuasse ordenado. Dependendo da ordem que os elementos forem inseridos pode se tornar uma operação custosa, mas dessa forma as demais operações se tornam muito mais eficientes. O pseudocódigo 1 representa o algoritmo de inserção.

```
1:  ENTRADA: key: elemento a ser inserido, qnt: quantidade
2:
3:  Procure o elemento no array
4:  SE encontrou:
5:      Aumente sua contagem com "qnt"
6:  SENÃO:
7:      SE o array está cheio:
8:          Aloque um novo array com o dobro de capacidade
9:          Copie os elementos do array antigo para o novo, inserindo o novo elemento
10:     SENÃO:
11:         ENQUANTO Os elementos, a partir da última posição, forem maiores que o
12:                  item a ser inserido
13:             Copie o elemento para a posição posterior
14:
15:         Insira o item
```

Pseudocódigo 1

Como podem existir elementos repetidos, a primeira operação do algoritmo é procurar (utilizando uma busca binária) se o multiconjunto já contém o elemento. Em caso afirmativo, basta aumentar o seu contador com a quantidade a ser inserida. Caso contrário, é necessário verificar se a memória ocupada pelo *array* já está totalmente preenchida. Se o multiconjunto estiver cheio, um novo *array* é alocado, os itens são copiados e o novo elemento é inserido já na sua posição. Se estiver vazio, é necessário arrastar todos os itens maiores até que que o novo elemento possa ser inserido.

- No melhor caso, o elemento é encontrado na primeira iteração da busca, bastando incrementar seu contador, o que resulta em uma complexidade pertencente à O(1).
- No pior caso, o elemento é menor que todos os itens do *array*. Assim, a busca realiza log(n) iterações, e todos os elementos são deslocados, gastando n comparações. Logo, a complexidade pertence à $O(\log(n) + n) \in O(n)$.

2.1.2 - Pertence

Como o array é mantido ordenado, basta realizar uma busca binária para verificar se um dado elemento está contido no multiconjunto. Logo, no melhor caso a complexidade pertence à O(1) e, no pior, à O(log(n)).

2.1.3 - Frequência

Como a frequência de um item é apenas um contador armazenado junto ao elemento, é feita uma busca binária para encontrá-lo e retornar o valor de seu contador. Assim como na operação anterior, a complexidade no melhor caso pertence à O(1) e, no pior, à O(log(n)).

2.1.4 - Remove

Para a operação de remoção, é necessário encontrar o elemento no *array*. Como os elementos são ordenados, basta fazer uma busca binária. Entretanto, para efetivamente removê-lo, é necessário deslocar os elementos maiores, de forma a mantê-los contínuos na memória.

- No melhor caso, o elemento está na última posição do *array*. Apesar da busca encontrá-lo somente após log(n) iterações, basta removê-lo sem a necessidade de deslocar nenhum item. Assim, a complexidade pertence à O(log(n)).
- No pior caso, assim como na inserção, o elemento é menor que todos os itens do *array*. A busca realiza log(n) iterações, e todos os elementos são deslocados (todos com exceção do item a ser removido), gastando n - 1 comparações. Logo, a complexidade pertence à $O(\log(n) + n) \in O(n)$.

2.1.5 - Apaga

Como os elementos repetidos são registrados pelo contador de cada item, a análise da operação “Apaga” é a mesma da “Remove”. Assim, no melhor caso, a complexidade pertence à O(log(*n*)) e, no pior, à O(*n*).

2.1.6 - União

O pseudocódigo 2 ilustra a implementação da operação de união.

1: **ENTRADA**:
2: ms1, ms2: multiconjuntos com os quais será realizada a operação
3:
4: *i* = posição inicial de ms1
5: *j* = posição inicial de ms2
6:
7: **ENQUANTO** *i* < tamanho de ms1 **E** *j* < tamanho de ms2:
8: *a* = *i*-ésimo item de ms1
9: *b* = *j*-ésimo item de ms2
10: **SE** *a* < *b*:
11: O item *a* pertence à união
12: Incremente o valor de *i*
13: **SENÃO, SE** *a* > *b*:
14: O item *b* pertence à união
15: Incremente o valor de *j*
16: **SENÃO**:
16: O item *a* é adicionado com quantidade igual a maior quantidade entre *a* e *b*
17: Incremente o valor de *i*
18: Incremente o valor de *j*
19:

20: **ENQUANTO** *i* < tamanho de ms1:

21: O item na posição i de ms1 é adicionado à união

22: Incremente o valor de *i*

23:

24: **ENQUANTO** j < tamanho de ms2:

25: O item na posição j de ms2 é adicionado à união

26: Incremente o valor de j

Pseudocódigo 2

Como os *arrays* são ordenados, é possível implementar a operação de união apenas iterando sobre eles. Os itens diferentes são sempre adicionados à união e, quando os itens pertencem à ambos os multiconjuntos, basta adicionar o maior número de ocorrências com o qual eles aparecem. Caso seus tamanhos sejam diferentes, é necessário adicionar o excedente. Assim, temos:

- No melhor caso um dos *arrays* é vazio, o que faz com que o algoritmo apenas itere sobre o multiconjunto que possua elementos adicionando-os à união. Daí sua complexidade pertence à O(n).
- No pior caso os elementos do primeiro *array* são menores que todos os elementos do segundo (exceto o último, que é maior que todos os itens do segundo *array*). Dessa forma o algoritmo fará a primeira comparação do primeiro *loop* n - 1 vezes (onde n é o tamanho do primeiro *array*) e a segunda comparação m vezes (onde m é o tamanho do segundo *array*). Logo, sua complexidade pertence à O($n + m$).

2.1.7 - Interseção

A operação de interseção segue o mesmo raciocínio da operação de união, sem o segundo e terceiro *loops* do Pseudocódigo 2. Entretanto, o elemento só é adicionado ao multiconjunto resultante caso a primeira e a segunda condição falhem, ou seja, quando o item na posição *i* do primeiro *array* é igual ao item na posição *j* do segundo. Assim, a complexidade do pior caso é analisada da mesma forma que na união. O melhor caso, apesar de ocorrer também quando um dos multiconjuntos é vazio, possui complexidade pertencente à O(1), pois a operação básica não ocorre nem uma vez.

2.1.8 - Diferença

A diferença, assim como a interseção, difere da união pela condição em que os itens são adicionados ao resultado e pelos últimos *loops* do Pseudocódigo 2 (são necessários apenas o primeiro e segundo laços, pois os elementos resultantes do segundo multiconjunto não pertencem à diferença). Os elementos são adicionados quando a primeira condição é satisfeita ou quando o item na posição *i* do primeiro *array* é igual ao item na posição *j* do segundo e sua quantidade é maior. Dessa forma, no melhor caso, o primeiro multiconjunto é vazio, resultando em uma complexidade pertencente à O(1). O pior caso é semelhante ao pior caso da união e interseção, cujas complexidades pertencem à O($n + m$).

2.1.9 - Espaço

Cada item do ODA possui o elemento e seu contador. Quando o *array* está cheio, ele aloca o dobro de memória. Caso ele possua $2^n + 1$ elementos, ele terá alocado 2^{n+1} posições na memória, ou seja, 2^n - 1 posições estarão desperdiçadas.

2.2 - Operações da RBT

As operações da árvore vermelho-preto foram baseadas em [2]. As implementações sofreram algumas alterações para se adaptarem ao problema, mas tanto a ideia quanto a complexidade são semelhantes.

2.2.1 - Insere

A operação de inserção altera a estrutura de dados, fazendo com que sejam necessárias algumas funções auxiliares para manter as propriedades e eficiência da árvore. Essas operações não serão discutidas aqui, basta saber que, no pior caso, sua complexidade pertence à O(log(n)). Para mais informações consulte [2].

Para inserir um novo elemento, primeiro é necessário verificar se a estrutura está vazia. Se estiver, adiciona-se o item como raiz da árvore. Caso contrário, é necessário buscar a posição em que deve ser inserido. Como se trata de um multiconjunto, pode ser que o elemento já esteja presente na estrutura. Nesse caso, basta incrementar seu contador com a quantidade a ser inserida. Se não estiver, um novo nó é criado com as informações do item, e a árvore é balanceada. Em média, a complexidade de inserção pertence à O(log(n)).

2.2.2 - Pertence

Para verificar se um elemento pertence ao multiconjunto, basta percorrer a árvore comparando se o nó atual é maior ou menor que o item que se deseja pesquisar. Se for, procura-se na subárvore esquerda, senão, procura-se na subárvore direita. Em média, a complexidade pertence à O(log(*n*)).

2.2.3 - Frequência

Análoga à "pertence", mas quando o elemento é encontrado, retorna o seu contador. Em média a complexidade pertence à O(log(*n*)).

2.2.4 - Remove

A operação de remoção, assim como a inserção, altera a estrutura de dados. Para manter as propriedades e eficiência da estrutura, são utilizadas operações de balanceamento que também não serão discutidas nesse relatório (consulte [2] para mais informações). Primeiro a existência do elemento na árvore é verificada. Caso exista, o nó é removido e de acordo com seu número de filhos, diferentes estratégias de balanceamento são aplicadas. Em média, a complexidade pertence à O(log(*n*)).

2.2.5 - Apaga

Semelhante à operação anterior, uma vez que a frequência dos elementos é mantida por um contador em cada nó. Assim, no caso médio, a complexidade pertence à O(log(*n*)).

2.2.6 - União, Interseção e Diferença

Os algoritmos de união, interseção e diferença da RBT são análogos aos do ODA. A diferença está apenas no modo em que os elementos são acessados. Dois ponteiros auxiliares são utilizados para acessar o menor elemento de cada multiconjunto, gastando log(*n*) e log(*m*) operações, respectivamente (onde *n* é o número de elementos distintos da primeira árvore, e *m* o número de elementos distintos da segunda). Em seguida, utilizando a função sucessor, que no pior caso pertence à O(log(*n*)), os próximos elementos são acessados de acordo com a necessidade dos algoritmos. Assim, a complexidade dessas operações no caso médio pertence, à O(*n*(log(*m*)) + *m*(log(*n*))), visto que para cada elemento de cada multiconjunto, a função sucessor pode gastar log(*n*) operações.

2.2.9 - Memória

Cada nó da árvore possui o elemento, uma *flag* para definir sua cor, um contador para elementos repetidos, e três ponteiros (para o pai, filho esquerdo e filho direito). Um nó só é alocado quando o item não existia previamente na árvore. Assim, o espaço ocupado pela estrutura é dado pelo número de elementos diferentes multiplicado pelo tamanho do nó.

2.3 – Comparação entre ODA e RBT

Por ser mantido ordenado, o ODA consegue grande eficiência em operações de consulta, pois pode utilizar busca binária para tal. A ordem de inserção pode tornar o acréscimo de elementos bastante ineficiente, uma vez que a cada item adicionado vários podem ter que ser deslocados para manter o *array* em ordem. Outra vantagem do ODA é a localidade de *cache* proporcionada pela implementação, visto que os itens são mantidos de forma contínua na memória. Entretanto, como dito na seção 2.1.9, o consumo de espaço pode ser custoso.

A RBT por sua vez, consegue manter uma complexidade igual para todos os casos de suas operações devido ao fato de ser implementada como uma árvore. Apesar de não possuir o melhor caso mais eficiente que o do ODA, sua complexidade ainda é bem baixa e, para uso na prática se torna mais eficiente, como mostrado na seção 3. A localidade de cache é perdida nessa abordagem, pois os elementos são conectados por ponteiros. Apesar de possuir nós maiores que os da ODA, o consumo de memória pode ser menor, visto que só são alocados nós quando realmente necessário.

A tabela 1 mostra as complexidades analisadas nas seções 2.1 e 2.2. Note que os melhores e piores casos da RBT são iguais. O ODA por sua vez, possui, em geral, um melhor caso muito bom e um pior caso ruim.

Operação	ODA		RBT	
	Melhor caso	Pior caso	Melhor caso	Pior caso
Insere	O(1)	O(n)	O(log(n))	O(log(n))
Pertence	O(1)	O(log(n))	O(log(n))	O(log(n))
Frequência	O(1)	O(log(n))	O(log(n))	O(log(n))
Remove	O(log(n))	O(n)	O(log(n))	O(log(n))
Apaga	O(log(n))	O(n)	O(log(n))	O(log(n))
União	O(n)	O($n + m$)	O(n(log(m)) + m(log(n)))	O(n(log(m)) + m(log(n)))
Interseção	O(1)	O($n + m$)	O(n(log(m)) + m(log(n)))	O(n(log(m)) + m(log(n)))
Diferença	O(1)	O($n + m$)	O(n(log(m)) + m(log(n)))	O(n(log(m)) + m(log(n)))

Tabela 1 – Comparação das análises de complexidades entre ODA e RBT. n e m são o número de elementos distintos do primeiro e segundo multiconjuntos.

3 - Análise experimental

A análise experimental foi dividida em dois casos. No primeiro foi testada a eficiência das estruturas para manipulação dos dados, isto é, para inserções, remoções e consultas. No segundo foi testada a eficiência das operações entre multiconjuntos (união, interseção e diferença). Os testes foram executados em um computador com as seguintes especificações:

- Compilador: MinGW g++ 7.2 (utilizando *flag* –O3)
- Sistema Operacional: Windows 10 Pro 64-bit (10.0, build 17134)
- Processador: AMD Ryzen 5 1600 - 3.2GHz - 6 cores - 19MB de cache
- Memória: 16GB RAM - DDR4 - 2666 MHz

A Tabela 2 mostra o tempo gasto para as operações do primeiro caso. Foram testadas inserções de dados ordenados de forma crescente, decrescente e aleatórios, consultas aleatórias e remoções aleatórias. Foram realizados testes com 1000, 10000, 100000 e 1000000 elementos.

A Tabela 3 contém os resultados para as operações de união, interseção e diferença. Cada função foi testada com 3 possibilidades de multiconjuntos: 2 completamente diferentes, 2 exatamente iguais e 2 com elementos aleatórios. Cada possibilidade foi testada para 1000, 10000, 100000 e 1000000 elementos.

Para os testes com aleatoriedade, foram gerados elementos utilizando a função *rand* da biblioteca padrão do C++ e armazenados em um vetor, para que as diferentes implementações fossem testadas com os mesmos dados, a fim de manter a comparação justa. STL é a implementação do multiconjunto da Standard Template Library do C++ (std::multiset).

Quantidade de elementos	Implementação	Tempo gasto pela operação (nanosegundos)				
		Inserção em ordem crescente	Inserção em ordem decrescente	Inserção de dados aleatórios	Consultas de dados aleatórios	Remoções de dados aleatórios
1 000	ODA	37190	470963	332464	61876	65402
	RBT	153247	120546	114775	36228	69571
	STL	102592	76303	138820	62838	69570
10 000	ODA	428003	44021764	17299966	802465	3781812
	RBT	1163142	906340	1540169	660760	1255154
	STL	924614	878127	1652699	1031053	1276314
100 000	ODA	4159801	4438554998	227037974	9185228	86375439
	RBT	12214268	10720908	14284070	10475006	16371505
	STL	9812003	10615109	24255090	21565567	22642146
1 000 000	ODA	46541048	908152871033	348547788	102891856	178941368
	RBT	136437652	120935938	126280041	100578397	139792423
	STL	113813139	121757639	800112146	756076597	203879661

Tabela 2 – Resultados para as operações de manipulação de dados

Quantidade de elementos	Relação entre os multiconjuntos	Implementação	Tempo gasto pela operação (nanosegundos)		
			União	Interseção	Diferença
1 000	Multiconjuntos completamente diferentes	ODA	74379	1282	24045
		RBT	261290	6421	120546
		STL	23404	7054	11221
	Multiconjuntos iguais	ODA	22122	21801	1604
		RBT	149721	131767	9618
		STL	16671	17633	11221
	Multiconjuntos com elementos aleatórios	ODA	20519	22443	1603
		RBT	154851	136576	15068
		STL	20519	21159	15709
10 000	Multiconjuntos completamente diferentes	ODA	873639	9297	321563
		RBT	3078414	57388	1324083
		STL	322845	70853	83036
	Multiconjuntos iguais	ODA	403958	452369	12504
		RBT	1458094	1419623	90730
		STL	179857	168957	106760
	Multiconjuntos com elementos aleatórios	ODA	351699	287900	12503
		RBT	1379548	1319595	155491
		STL	232756	294312	210955
100 000	Multiconjuntos completamente diferentes	ODA	9060193	128561	4163969
		RBT	38440095	2201249	16818423
		STL	5259785	2283643	2562566
	Multiconjuntos iguais	ODA	4480723	3968402	207429
		RBT	19282886	18359233	3439410
		STL	4254700	4161725	3717052
	Multiconjuntos com elementos aleatórios	ODA	1109601	834204	59952
		RBT	6198185	5950680	1610380
		STL	8331466	7923340	7236933
1 000 000	Multiconjuntos iguais	ODA	101238195	1245536	48191824
		RBT	404442379	21571979	185089219
		STL	51242346	22608162	25880860
	Multiconjuntos completamente diferentes	ODA	50653401	48344751	1905334
		RBT	213282255	196987374	33419479
		STL	40975089	41666947	35928825
	Multiconjuntos com elementos aleatórios	ODA	1181416	868188	65403
		RBT	6963139	6348226	1726118
		STL	130599502	129732597	126908739

Tabela 3 – Testes para operações entre multiconjuntos

4 – Conclusões

A RBT e STL possuem tempos bem próximos para os testes de manipulações de dados, pois ambas são implementações da árvore vermelho-preto.

Os testes condizem com a análise teórica. Note que no melhor caso de inserção do ODA (inserção em ordem crescente), o tempo é sempre menor que para a RBT e STL. Já no pior caso (inserção em ordem decrescente) o tempo é bem pior. Quanto a inserção de dados aleatórios, que seria o caso mais próximo da realidade, o tempo da ODA é também pior que das demais estruturas.

Os tempos de consultas são parecidos, sendo que a STL perde em todas as quantidades de elementos. Para remoções, em alguns casos o ODA possui um tempo consideravelmente maior que as demais implementações, pois dependendo do item removido, vários elementos podem ser deslocados.

Para as operações entre multiconjuntos, os tempos da ODA são bem menores que os da RBT, confirmando a análise teórica.

Podemos concluir que cada abordagem pode ser melhor para atacar determinado tipo de problema, dependendo da operação mais frequente para a solução.

5 - Referências

[1] – DSnA: Multiset implementation. Disponível em: <https://github.com/marcelodmmenezes/DSnA/tree/master/Multiset>

[2] – CORMEN, T. et al. Introduction to Algorithms. 3 ed. The MIT Press, 2009.