Uma proposta de Range min-Max tree k-ária para consultas sobre árvores sucintas

Defesa de TCC

Danyelle da Silva Oliveira Angelo

Orientador: Prof. Me. Daniel Saad Nogueira Nunes

Banca examinadora: Prof. Me. João Victor de Araujo Oliveira e
Prof. Dr. Felipe Alves da Louza

Instituto Federal de Brasília, Câmpus Taguatinga

14 de setembro de 2021

Sumário

Sumário

Dados nunca dormem

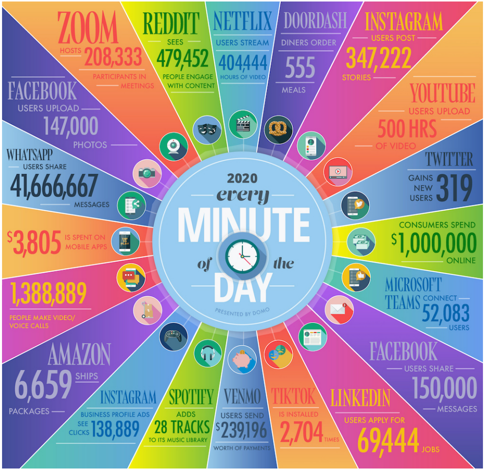


Figura: Infográfico: Data Never Sleeps 8.0

Aumento na produção de dados

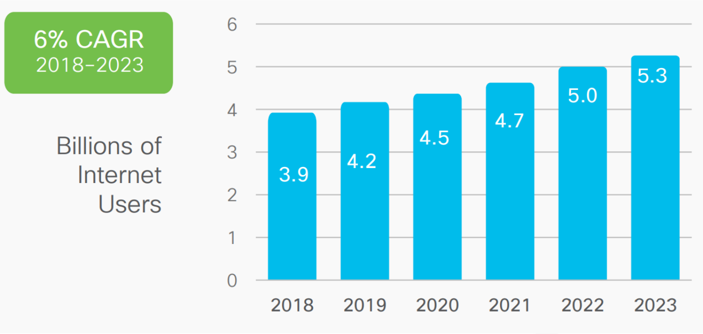


Figura: Número de usuários conectados à internet

Fonte: Cisco [2020]

Aumento na produção de dados

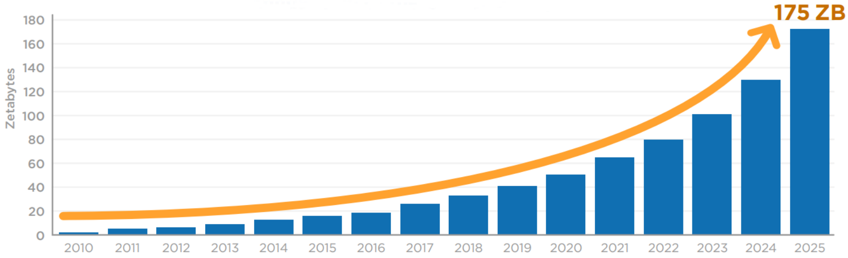


Figura: Esfera global de dados por ano

Fonte: Reinsel et al. [2018]

Segmentação da indústria e Gargalo de Von-Neumann

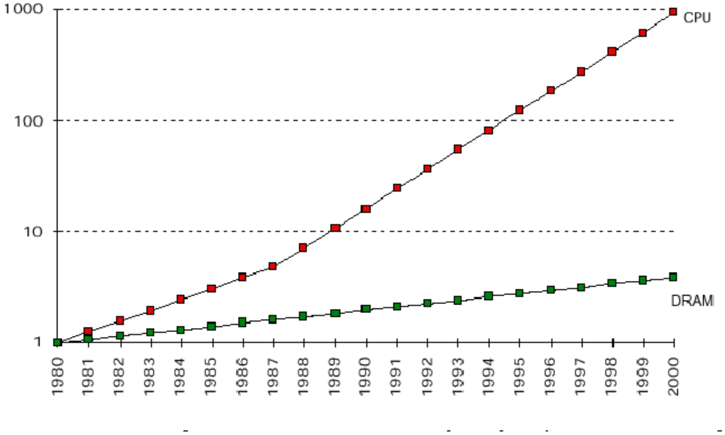


Figura: Lacuna de desempenho entre processador e memória

Fonte: Patterson et al. [1997]

Hierarquia de memória

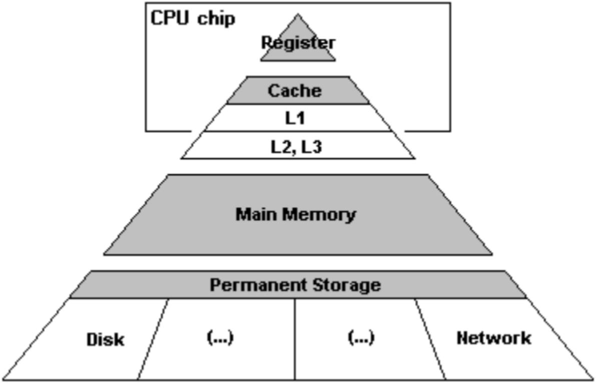


Figura: Hierarquia de memória

Fonte: Carvalho [2002]

Hierarquia de memória

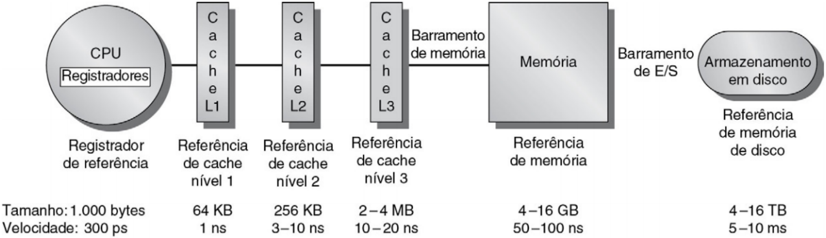


Figura: Hierarquia de memória de um servidor

Retirado de: Hennessy and Patterson [2012]

Solução

- Atuar nos níveis com menor latência;
- Estrutura de dados e compactação de dados:
  - Representação do DNA através de árvore de sufixos;
  - Compressão de dados clássica vs Estrutura de dados sucintas.

Proposta

- Sadakane and Navarro [2010], range min-Max tree (rmM-tree);
- Um novo gargalo;
- range min-Max k-ária.

Sumário

Estrutura de dados sucintas

Estrutura de dados sucintas são uma forma de compressão de dados, e de acordo com Navarro [2016], estas propiciam:

- Representação dos objetos obedecendo o limite da entropia da informação;
- Operações eficientes em questões de tempo e espaço;
- Manipulação de dados em dispositivos com memória limitada;

Vetores de bits - access

Sequência de n elementos sobre o alfabeto $\Sigma = \{0, 1\}$, no qual podem ser realizadas as seguintes operações [Navarro, 2016]:

- $access(BV, i)$: retorna o i-ésimo bit do vetor BV, com $0 \leq i < n$;

  Exemplo: $access(BV, 10)$

BV =	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Vetores de bits - access

Exemplo: $access(BV, 10) = 1$

BV =	1	0	1	0	0	1	0	1	1	0	**1**	1	1	0
	0	1	2	3	4	5	6	7	8	9	**10**	11	12	13

Vetores de bits - rank

Sequência de n elementos sobre o alfabeto $\Sigma = \{0, 1\}$, no qual podem ser realizadas as seguintes operações [Navarro, 2016]:

- $rank_v(BV, i)$: seja $v \in \{0, 1\}$, e $0 \leq i < n$, retorna o número de ocorrências de v no intervalo $BV[0, i]$.

  Exemplo: $rank_0(BV, 8)$

BV =	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Vetores de bits - rank

Exemplo: $rank_0(BV, 8) = 4$

BV = 1 0 1 0 0 1 0 1 1 0 1 1 1 0

BV =	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Vetores de bits - select

Sequência de n elementos sobre o alfabeto $\Sigma = \{0, 1\}$, no qual podem ser realizadas as seguintes operações [Navarro, 2016]:

- $select_v(BV, i)$: dado $v \in \{0, 1\}$ e $i \geq 0$, retorna a posição do i-ésimo bit v em $BV[0, n-1]$.

  Exemplo: $select_1(BV, 7)$

BV =	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Vetores de bits - select

Exemplo: $select_1(BV, 7) = 12$

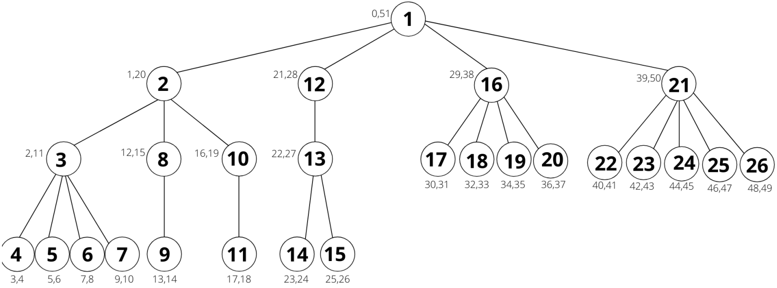
BV =	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
	0	1	**2**	3	4	**5**	6	**7**	**8**	9	**10**	**11**	**12**	13

Representações sucintas de árvores

- Parênteses balanceados (BP);
- Depth-First Unary Degree Sequence (DFUDS);
- Level-order Unary Degree Sequence (LOUDS);

Representações sucintas de árvores

Representação de árvores sucintas via Parênteses Balanceados



(((( ) ( ) ( ) ( )) (( )) (( ))) ((( ) ( ))) (( ) ( ) ( ) ( )) (( ) ( ) ( ) ( ) ( )))

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Representações sucintas de árvores

Representação de árvores sucintas via parênteses balanceados

- Sequência de $2n$ parênteses balanceados;
- Complexidade de espaço $2n + o(n)$ bits;
- Através de estruturas auxiliares suporta:
  - findclose(B,i);
  - findopen(B,i);
  - excess(B,i);

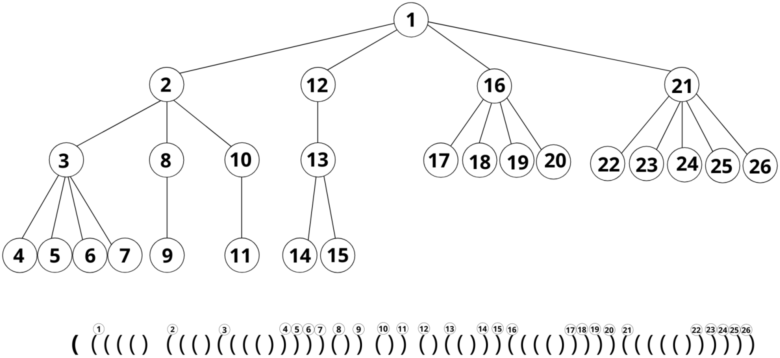
Representações sucintas de árvores

Exemplo: *excess(B,6) = -1*

BV =	1	0	1	0	0	1	0	1	1	0	1	1	1	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

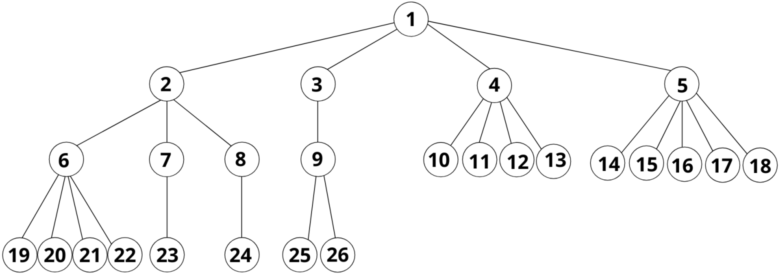
Representações sucintas de árvores

Representação de árvores sucintas via DFUDS



Representações sucintas de árvores

Representação de árvores sucintas via LOUDS



((((() ((() () (((() ((((() (((() () () (())))))))))))))))))

range min-Max tree (rmM-tree)

- Construção bottom-up;
- Árvore binária completa, baseada em intervalos de tamanho b;
- Cada nó cobre valores de excessos dentro de um intervalo;
- Complexidade de espaço igual à $n + O(\frac{n}{b} \log n)$ bits;
- Operações realizadas em tempo $O(\log n)$.

range min-Max tree binária

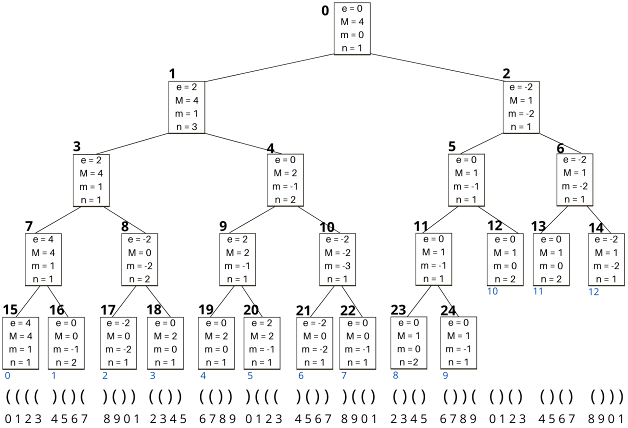


Figura: rmM-tree binária com blocos de tamanho 4

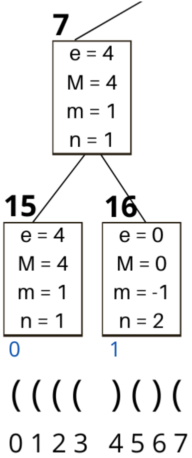
rmM-tree: Registros

Valores de excesso Dado um nó v que cobre um intervalo $BP[s, e]$, então:

- $R[v].e$: excesso total no intervalo
  $R[v].e = excess(e) - excess(s-1)$.
- $R[v].M$: excesso máximo no intervalo
  $R[v].M = \max\{excess(i) - excess(s-1) | s \leq i \leq e\}$.
- $R[v].m$: excesso mínimo no intervalo
  $R[v].m = \min\{excess(i) - excess(s-1) | s \leq i \leq e\}$.
- $R[v].n$: número de vezes que o excesso mínimo ocorre no intervalo
  $R[v].n = |\{BP[i] = R[v].m | s \leq i \leq e\}|$.

rmM-tree: Registros

Nós internos e raíz



R[7].e = R[15].e + R[16].e.
R[7].M=max(R[15].M, R[15].e+ R[16].M).
R[7].m=min(R[15].m, R[15].e+ R[16].m).
R[7].n= R[15].n.

rmM-tree: Operações

Operações		
fwdsearch(i,d)	bwdsearch(i,d)	minExcess(i,j)
maxExcess(i,j)	minSelectExcess(i,j,t)	minCount(i,j)
enclose(i)	rank_v(i)	select_v(i)
findClose(i)	findOpen(i)	rmq(i,j)
inspect(i)	preRank(i)	postRank(i)
preSelect(i)	postSelect(i)	isLeaf(i)
isAncestor(i,j)	depth(i)	parent(i)

rmM-tree: Operações

Operações		
firstChild(i)	lastChild(i)	nextSibling(i)
prevSibling(i)	subtreeSize(i)	levelAncestor(i,d)
level-next(i)	levelPrev(i)	levelLmost(d)
levelRmost(d)	lca(i,j)	deepestNode(i)
degree(i)	child(i,q)	childRank(i)
leafRank(i)	leafSelect(i)	lmostLeaf(i)

rmM-tree: operações

Problema: Dado um nó codificado em $i = 1$, encontrar o nó codificado em $j > i$, mais à esquerda de i.
Solução:

$$nextSibling(i) = findClose(i) + 1$$

$$findClose(i) = fwdSearch(i, -1)$$

rmM-tree: $nextSibling$

Problema: Computar $nextSibling(1)$.

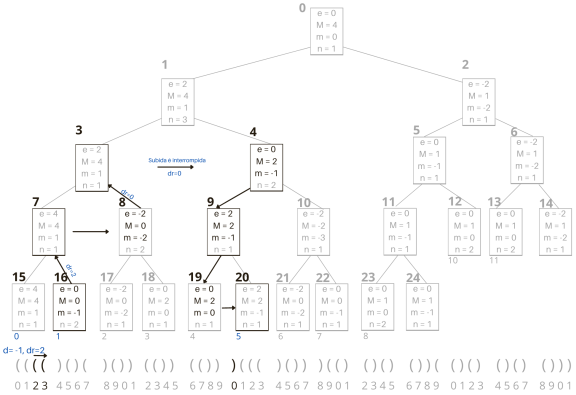


Figura: Simulação de $fwdSearch(1, -1) = 20$

rmM-tree: $nextSibling$

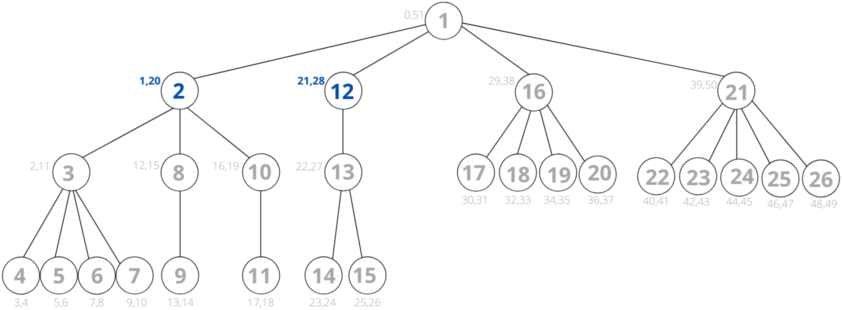
Problema: Dado um nó codificado em $i = 1$, encontrar o nó codificado em $j > i$, mais à esquerda de i.
Solução:

$$findClose(1) = fwdSearch(1, -1) = 20$$

$$nextSibling(1) = fwdSearch(1, -1) + 1 = 21$$

rmM-tree: $nextSibling$

Solução: $nextSibling(1)$.



(((( ) ( ) ( ) ( )) (( )) (( ))) ((( ) ( ))) (( ) ( ) ( ) ( )) (( ) ( ) ( ) ( ) ( )))

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Figura: Árvore de entrada

rmM-tree: operações

Problema: Verificar o ancestral comum mais baixo dos nós codificados em $i = 5$ e $j = 17$.

((((	)()(	)())	(())	(())	)(((	)())	)(()	()()	())(	()()	()()	()))
0123	4567	8901	2345	6789	0123	4567	8901	2345	6789	0123	4567	8901

Figura: Sequência de parênteses balanceados, representando uma árvore T

rmM-tree: lca

Problema: Verificar o ancestral comum mais baixo dos nós codificados em $i = 5$ e $j = 17$.
Solução:

$$lca(i,j) = \begin{cases} i, \text{se } isAncestor(i,j); \\ j, \text{se } isAncestor(j,i); \\ parent(rmq(i,j)+1) \end{cases}$$

rmM-tree: lca

Problema: Verificar o ancestral comum (mais baixo) dos nós codificados em $i = 5$ e $j = 17$.
Solução:

- *isAncestor(,i,j)* verifica se o nó codificado em i é ancestral do nó j;
- O mesmo vale para *isAncestor(j,i)*;
- Usar o terceiro caso.

rmM-tree: lca

Problema: Computar $lca(5, 17)$.
Solução: $parent(rmq(i, j) + 1)$

- $rmq(i, j) = fwdSearch(i - 1, minExcess(i, j))$;
- $parent(rmq(i, j) + 1) = bwdSearch(rmq(i, j) + 1, 0) + 1$.

rmM-tree: lca

Problema: Computar $lca(5, 17)$.

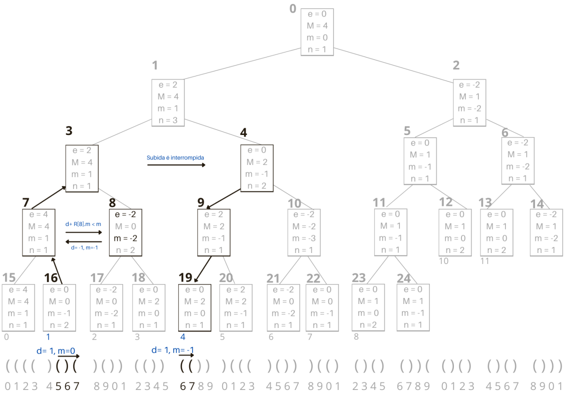


Figura: Simulação de $minExcess(5, 17) = -1$

rmM-tree: lca

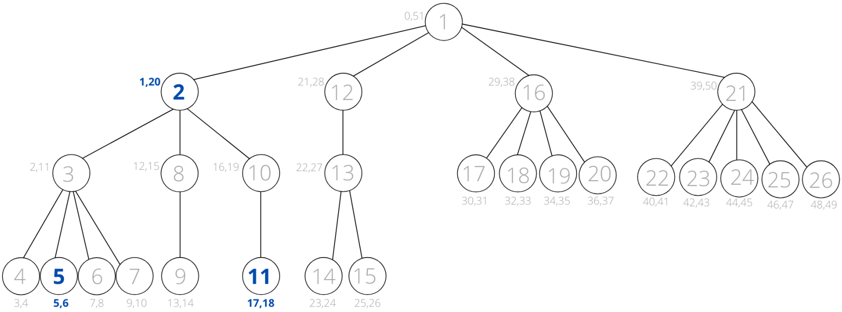
Solução: Computar $lca(5, 17)$.

((((	) ( ) (	) ( ))	(( ))	(( ))	) (((	) ( ))	) (( )	( ) ( )	( )) (	( ) ( )	( ) ( )	( )))
0 1 2 3	4 5 6 7	8 9 0 1	2 3 4 5	6 7 8 9	0 1 2 3	4 5 6 7	8 9 0 1	2 3 4 5	6 7 8 9	0 1 2 3	4 5 6 7	8 9 0 1

Figura: Nós acessados durante a operação $lca(5, 17)$

rmM-tree: lca

Solução: Computar $lca(5, 17)$.



(**(** ((( ) **(**) ( ) ( )) (( )) (**(**))) ((( ) ( ))) (( ) ( ) ( ) ( )) (( ) ( ) ( ) ( ) ( )))

0 **1** 2 3 4 **5** 6 7 8 9 0 1 2 3 4 5 6 **7** 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Figura: Árvore de entrada

Aproveitamento de cache

- Expansão da memória principal;
- Dados residindo em memória principal: um novo gargalo;
- Falhas de cache.

Aproveitamento de cache

- Maximização da quantidade de informação em um nó Hankins [2003]:
  - Altura da árvore;
  - Linha de cache.
- Fator de ramificação Rao and Ross [2000]:
  - Cache Sensitive Tree (CSS-tree);
  - Cache Sensitive B$^+$-Tree (CSB^+-tree);
  - Árvores B$^+$.

Sumário

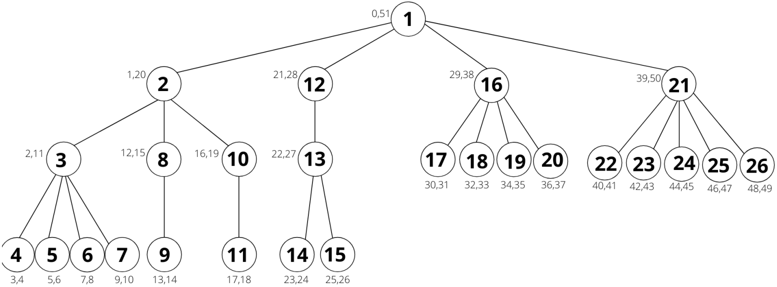
range min-Max tree k-ária (rmM-tree k-ária)

Características:

- Alto fator de ramificação;
- Maior cobertura de área por nó;
- Cada nó cobre até k intervalos;
- Mesmas definições de registros da estrutura binária;
- Complexidade de tempo e espaço eficientes, usando os mesmos campos defindos por Navarro [2016] em sua estrutura.

rmM-tree k-ária

Árvore de entrada



(((( ) ( ) ( ) ( )) (( )) (( ))) ((( ) ( ))) (( ) ( ) ( ) ( )) (( ) ( ) ( ) ( ) ( )))

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

rmM-tree k-ária: Registros

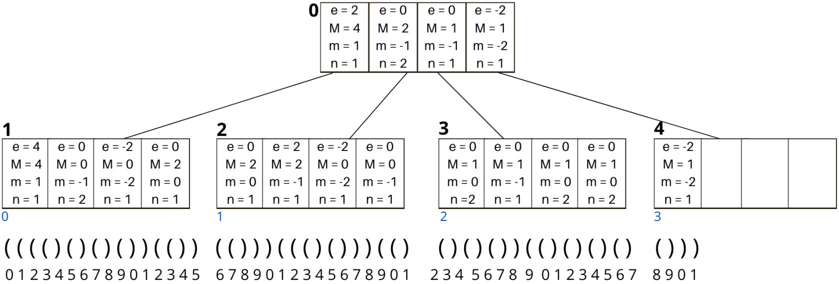
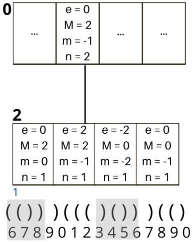


Figura: rmM-tree 4-ária com blocos de tamanho 4

rmM-tree k-ária: Registros

Nós internos e raíz



$$
\begin{aligned}
R[0][1].M = & max(R[2][0].M, \\
& R[2][0].e + R[2][1].M, \\
& R[2][0].e + R[2][1].e + R[2][2].M, \\
& R[2][0].e + R[2][1].e + R[2][2].e + \\
& R[2][3].M) \\
& = max(2, 2, 2, 0) = 2;
\end{aligned}
$$

rmM-tree k-ary: Operações

Tabela: Operações suportadas pela rmM-tree binária e rmM-tree-kária

Operação	**rmM-tree binária**	**rmM-tree k-ária**
fwdSearch(i,d)	✓	✓
bwdSearch(i,d)	✓	✓
minExcess(i,j) / maxExcess(i,j)	✓	✗
minCount(i,j)	✓	✗
minSelectExcess(i,j,t)	✓	✗
enclose(i)	✓	✓
rmq(i,j) / rMq(i,j)	✓	✗
rank_1(i) / rank_0(i)	✓	✓
select_1(i) / select_0(i)	✓	✓

rmM-tree k-ary: Operações

Tabela: Operações suportadas pela rmM-tree binária e rmM-tree-kária

Operação	rmM-tree binária	rmM-tree k-ária
preRank(i)/postRank(i)	✓	✓
preSelect(i)/postSelect(i)	✓	✓
isLeaf(i)	✓	✓
isAncestor(i,j)	✓	✓
depth(i)	✓	✓
parent(i)	✓	✓
firstChild(i) / lastChild(i)	✓	✓
child(i,t)	✓	✗
nextSibling(i) / prevSibling(i)	✓	✓

rmM-tree k-ary: Operações

Tabela: Operações suportadas pela rmM-tree binária e rmM-tree-kária

Operação	**rmM-tree binária**	**rmM-tree k-ária**
subtreeSize(i)	✓	✓
levelAncestor(i,d)	✓	✓
levelNext(i) / levelPrev(i)	✓	✓
levelLeftMost(d) / levelRightMost(d)	✓	✓
lca(i,j)	✓	✗
deepestNode(i)	✓	✗
degree(i)	✓	✗
childRank(i)	✓	✗
leafRank(i)/leafSelect(i)	✓	✓
leftMostLeaf(i)/rightMostLeaf(i)	✓	✓

rmM-tree k-ary: operações

Problema: Dado um nó codificado em $i = 1$, encontrar o nó codificado em $j > i$, mais à esquerda de i.
Solução:

$$nextSibling(i) = findClose(i) + 1$$

$$findClose(i) = fwdSearch(i, -1)$$

rmM-tree k-ary: $nextSibling$

Problema: Computar $nextSibling(1)$.

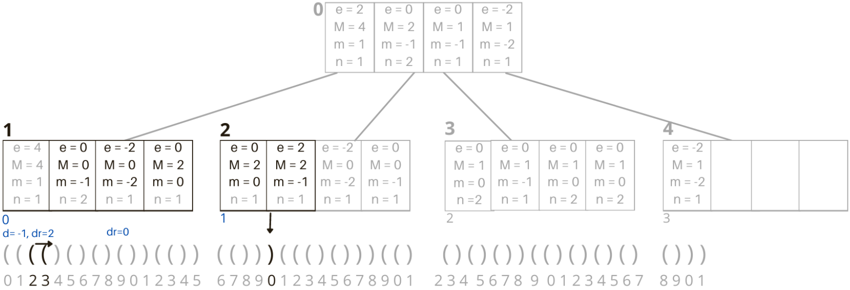


Figura: Simulação de $fwdSearch(1, -1) = 20$ em uma rmM-tree 4-ária

rmM-tree k-ary: $nextSibling$

Problema: Computar $nextSibling(1)$.

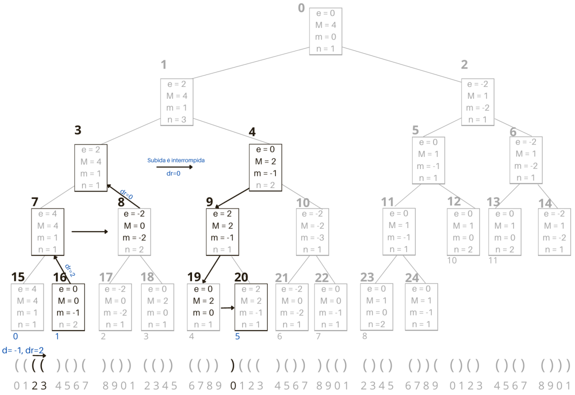


Figura: Simulação de $fwdSearch(1, -1) = 20$ usando a rmM-tree binária

rmM-tree k-ary: $nextSibling$

Problema: Dado um nó codificado em $i = 1$, encontrar o nó codificado em $j > i$, mais à esquerda de i.
Solução:

$$findClose(1) = fwdSearch(1, -1) = 20$$

$$nextSibling(1) = fwdSearch(1, -1) + 1 = 21$$

rmM-tree k-ary: $nextSibling$

Solução: $nextSibling(1)$.

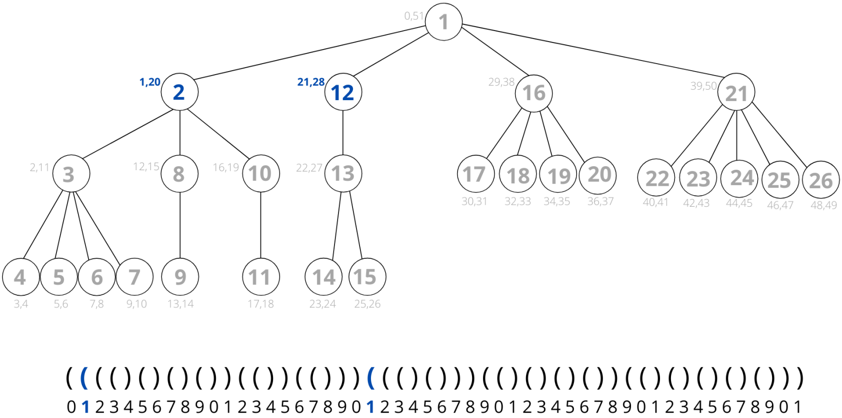


Figura: Árvore de entrada

Sumário

Hardware

- Arquitetura: x86
- Processador: Intel Xeon Gold 5120
- Frequência máxima: 3,20 GHz
- Threads por core: 2
- Cores: 28
- Cache L1: 896 KiB
- Cache L2: 28 MiB
- Cache L3: 38.5 MiB
- Memória RAM total: 527,03 Gb

Base de dados

Tabela: Conjunto de dados usados nos testes experimetais, retirados de Fuentes [2016].

Conjunto de dados	**Tamanho (MB)**	**Quantidade de parênteses**	**Tamanho da árvore representada**
Complete tree (ctree)	18	2.147.483.644	1.073.741.822
DNA	135	1.154.482.174	577.241.087
Proteins (prot)	82	670.721.006	335.36203
Wikipedia (wiki)	13	498.753.914	249.376.957

Experimentos

- Implementação: rmM-tree binária e rmM-tree k-ária;
- Validação das respostas;
- Testes de desempenho.

Experimentos: validação

```
TEST_F(RMMTreeFixtureTest, bwdSearch_findOpen){
    for(int i=0;i<argsFindOpen.size();i++){
        EXPECT_EQ(t->findOpen(argsFindOpen[i]),bps->find_open(argsFindOpen[i])) << "Resposta errada ao cal
    }
}

TEST_F(RMMTreeFixtureTest, bwdSearch_enclose){
    int k=0;
    for(int i=0;i<(t->size)/2;i++){
        k = rand()%(t->size);
        EXPECT_EQ(t->enclose(k),bps->enclose(k)) << "Resposta errada ao calcular o enclose de i=" << k;
    }
}
```

Figura: Testes unitários para as operações *findopen* e *enclose*

Experimentos: desempenho

```
static void BM_Parent_k(benchmark::State& st){
    for(auto _ :st){
        for(int i=0; i < args_par_openII.size();i++){
            t->parent(args_par_openII[i]);
        }
    }
}
BENCHMARK(BM_Parent_k);

static void BM_SubTreeSize_k(benchmark::State& st){
    for(auto _ :st){
        for(int i=0; i < args_par_open.size();i++)
            t->subtreeSize(args_par_open[i]);
    }
}
BENCHMARK(BM_SubTreeSize_k);
```

Figura: Testes de desempenho para as operações *parent* e *subtreeSize*

Resultados

Tabela: Tempo (ns) médio de operações sobre o conjunto de dados Complete tree (ctree)

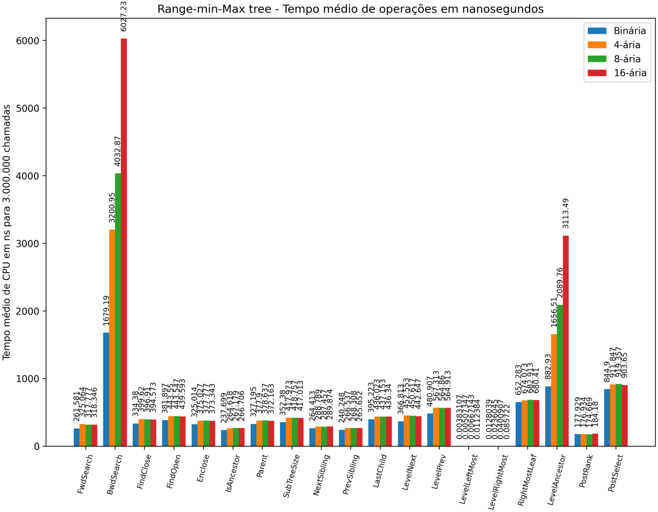
Operação	Binária	4-ária	8-ária	16-ária
fwdSearch	261,58	325,06	317,77	318,34
bwdSearch	167,19	3200,95	4032,87	6027,23
findClose	334.38	399,62	396,91	394,57
findOpen	381,89	443,55	443,54	439,59
enclose	325,01	375,02	377,77	373,34
isAncestor	237,69	264,61	267,17	266,70
parent	327,19	377,9	378,63	372,16
subTreeSize	352,38	417,97	418,75	417,01
nextSibling	264,41	288,8,78	287,45	289,87

Resultados

Tabela: Tempo (ns) médio de operações sobre o conjunto de dados Complete tree (ctree)

Operação	Binária	4-ária	8-ária	16-ária
prevSibling	240,24	266,53	268,30	265,65
lastChild	395,22	436,02	437,15	436,34
levelNext	366,81	452,55	450,61	442,64
levelPrev	480,90	567,31	564,86	564,91
rightMostLeaf	652,28	674,07	682,91	680,41
levelAncestor	882,93	1656,51	2089,76	3113,49
postRank	177,92	176,93	174,66	184,18
postSelect	844,9	911,84	919,35	372,16

Resultados


Range-min-Max tree - Tempo médio de operações em nanosegundos
Tempo médio de CPU em ns para 3.000.000 chamadas
Binária
4-ária
8-ária
16-ária
0
1000
2000
3000
4000
5000
6000
FwdSearch
BwdSearch
FindClose
FindOpen
Enclose
IsAncestor
Parent
SubTreeSize
NextSibling
PrevSibling
LastChild
LevelNext
LevelPrev
LevelLeftMost
LevelRightMost
RightMostLeaf
LevelAncestor
PostRank
PostSelect
261.581 325.064 317.777 318.346
1679.19 3200.95 4032.87 6027.23
334.38 399.62 396.91 394.573
381.897 443.55 443.547 439.593
325.014 375.027 377.777 373.343
237.699 264.618 267.179 266.706
327.195 377.9 378.637 372.163
352.38 417.973 418.757 417.013
264.413 288.789 287.457 289.874
240.248 266.537 268.308 265.652
395.223 436.023 437.153 436.34
366.813 452.553 450.617 442.647
480.907 567.313 564.86 564.913
0.00383107 0.00507347 0.00667143 0.0112384
0.0128039 0.0236045 0.0400907 0.085722
652.283 674.073 682.913 680.41
882.93 1656.51 2089.76 3113.49
177.929 176.934 174.669 184.18
844.9 911.847 919.357 903.65

Sumário

Trabalhos futuros

- Redução do tempo das operações da rmM-tree;
- Implementar demais operações de percurso para a rmM-tree k-ária;
- Monitorar uso da cache;
- Impacto da proposta em diferentes ambientes.

Referências

C. Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

Cisco. Cisco annual internet report (2018–2023). *Cisco*, 2020.

Domo. Data never sleeps 8.0. `https://www.domo.com/learn/infographic/data-never-sleeps-8`, 2020. Online; acesso em 05 de set. de 2021.

G. Fuentes, J. e Navarro. Parênteses balanceados. `http://www.inf.udec.cl/~jfuentess/datasets/parentheses.php`, 2016. Online; acesso em 24 de ago. de 2021.

J. M. Hankins, R. A. e Patel. Effect of node size on the performance of cache-conscious b+-trees. volume 29, pages 475–476. ACM SIGMETRICS international conference, 2003.

Referências

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, Waltham, MA, USA, fifth edition, 2012.

G. Navarro. *Compact data structures:a practical approach*. Sheridan Books, Inc, New York, NY, USA, 1 edition, 2016.

David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, I Thomas, and Katherine Yelick. A case for intelligent ram: Iram. 03 1997.

Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, page 475–486, New York, NY, USA, 2000. Association for Computing Machinery.

Referências

D. Reinsel, J. Gantz, and J. Rydning. The digitization of the world, from edge to core. *International Data Corporation (IDC)*, 1 2018.

K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, page 134–149, USA, 2010. Society for Industrial and Applied Mathematics.

Obrigada pela atenção!

Perguntas